Open Source Software Evolution and Its Dynamics

by

Jingwei Wu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2006

©Jingwei Wu 2006

I hereby declare that I am the sole author of this thesis. The is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis undertakes an empirical study of software evolution by analyzing open source software (OSS) systems. The main purpose is to aid in understanding OSS evolution. The work centers on collecting large quantities of structural data cost-effectively and analyzing such data to understand software evolution *dynamics* (the mechanisms and causes of change and growth).

We propose a multipurpose systematic approach to extracting program facts (*e.g.*, function calls). This approach is supported by a suite of C and C++ program extractors, which cover different steps in the program build process and handle both source and binary code. We present several heuristics to link facts extracted from individual files into a combined system model of reasonable accuracy. We extract historical sequences of system models to aid software evolution analysis.

We propose that software evolution can be viewed as *Punctuated Equilibrium* (*i.e.*, long periods of small changes interrupted occasionally by large avalanche changes). We develop two approaches to study such dynamical behavior. One approach uses the evolution spectrograph to visualize file level changes to the implemented system structure. The other approach relies on automated software clustering techniques to recover system design changes. We discuss lessons learned from using these approaches.

We present a new perspective on software evolution dynamics. From this perspective, an evolving software system responds to external events (*e.g.*, new functional requirements) according to *Self-Organized Criticality* (SOC). The SOC dynamics is characterized by the following: (1) the probability distribution of change sizes is a power law; and (2) the time series of change exhibits long range correlations with power law behavior. We present empirical evidence that SOC occurs in open source software systems.

Acknowledgements

First and foremost, I thank my advisor Prof. Richard C. Holt for his support, patience and encouragement throughout my doctoral studies. It is Richard who guided me through the maze of academic research and made the work in this thesis happen. His technical and editorial advice was essential to the completion of this thesis. Thanks Ric!

I thank my other dissertation committee members, Prof. Charlie Clarke, Prof. Michael Godfrey, Prof. Kostas Kontogiannis, and Prof. Gail Murphy, for their invaluable time and effort put into reading my thesis. In particular, I want to thank Michael for encouraging me to purse research in the field of software evolution and for providing constructive comments on many aspects of this thesis.

I thank Prof. Margaret-Anne Storey and Prof. Hausi Müller for showing me the door to exciting research in software engineering in my early graduate school life at the University of Victoria. My sincere gratitude goes to Margaret for her early supervision of my masters thesis. She taught me many valuable lessons on the workings of academic research.

I also thank many current and previous members of SWAG, in particular, Cory Kapser, Davor Svetinovic, Lijie Zou, Xinyi Dong, Yuan Lin and Ahmed E. Hassan (with whom I have collaborated on many research papers). I appreciate their great friendship as well as their insightful comments on many ideas in this thesis. They have made my life at SWAG a very enjoyable experience.

I feel honored to have been awarded by IBM the Center for Advanced Studies Fellowship for four years. I thank IBM for offering me opportunities to work on many exciting research projects at the IBM Toronto Lab during summers. In particular, I thank my mentors, Dr. Marin Litoiu and Dr. Kelly Lyons, for their inspiring guidance.

I am grateful to my fiancee, Hong, for sharing her experience of writing dissertation with me, for listening to my frustrations and for believing in me. I am greatly indebted to

my three elder sisters for looking after my mother wholeheartedly when I am thousands of miles away from home. Without their understanding and support, I would not have been able to come this far.

This thesis would not have been possible if it were not for a truly great person, my dear mom. Her love, encouragement and understanding are my greatest source of strength throughout this long journey.

I dedicate this thesis to my dear mom.

Jingwei Wu
in January 2006

Contents

1 Introduction **1**

1.1 Prior Research 1

1.1.1 Open Source Software Evolution 3

1.2 Problem Definition 5

1.3 Thesis Organization 6

1.4 Thesis Overview 8

1.5 Contributions 11

2 An Extractor Suite for C and C++ **15**

2.1 Introduction 15

2.2 The CX Suite 17

2.2.1 CPPX 18

2.2.2 BFX 19

2.2.3 LDX 21

2.2.4 CTSX 23

2.3 Applications 27

2.3.1 Creating Comprehension Pipelines 28

2.3.2 Building Software Evolution Database (EvolDB) 32

2.4 Discussion 37
2.4.1 Performance 37
2.4.2 Accuracy and Robustness 38
2.4.3 Systematic Support for C/C++ Extraction 40
2.5 Related Work 41
2.6 Conclusion 42

3 Improving Linkage Resolution in System Model Extraction 43
3.1 Introduction 44
3.2 Program Model Linking 46
3.2.1 Sample Schema of LPM 46
3.2.2 LPM Linking 48
3.2.3 Linkage Anomalies 48
3.3 Linking Rules and Heuristics 50
3.3.1 Linking Rules 50
3.3.2 Linking Heuristics 51
3.4 Linking Methods 54
3.4.1 Raw Linking 55
3.4.2 Heuristics-Based Linking 55
3.4.3 Simulation-Based Linking 57
3.4.4 Simulation plus Heuristics 61
3.4.5 Summary 62
3.5 Experimental Setup 63
3.5.1 Experimental Conditions 63
3.5.2 Baseline for Comparison 66
3.6 Experimental Results 67

3.6.1 Result Analysis Under Condition C1 67
3.6.2 Result Discussions Under All Conditions 71
3.7 Conclusion 73

4 An Empirical Study of Punctuated Software Evolution 75
4.1 Introduction 76
4.2 Punctuated Software Evolution 77
4.2.1 Software Architecture 78
4.2.2 Periods of Punctuation 78
4.2.3 Periods of Equilibrium 79
4.3 Methodology 79
4.3.1 Analysis Overview 80
4.3.2 Evolution Spectrograph 82
4.4 Case Studies 87
4.4.1 OpenSSH 88
4.4.2 PostgreSQL 91
4.4.3 Linux Kernel 94
4.5 Discussions 96
4.5.1 Threats to Validity 97
4.6 Related Work 98
4.6.1 Evolution Understanding 98
4.6.2 Evolution Visualization 99
4.7 Conclusion 101

5 Clustering Comparison in the Context of Software Evolution 103
5.1 Introduction 104

5.2 Target Systems 106
5.3 Experimental Design 107
5.4 A Simple Ordinal Measure 111
5.5 Empirical Results and Interpretation 112
5.5.1 Stability Comparison 112
5.5.2 Authoritativeness Comparison 117
5.5.3 Extremity Comparison 121
5.6 Discussion 125
5.7 Related Work 128
5.8 Conclusions 129

6 Fractal Nature of Software Evolution and SOC Dynamics 131

6.1 Introduction 132
6.2 Background 134
6.2.1 Fractal 134
6.2.2 Power Law 135
6.2.3 Time Series Analysis 136
6.2.4 Self-Organized Criticality 139
6.3 Data Collection 139
6.3.1 Software Change 139
6.3.2 Time Series of Change 142
6.4 Examining Fractals in Software Evolution 142
6.4.1 Power Law Distribution of Software Changes 142
6.4.2 Long Range Correlations in Time Series 148
6.4.3 Summary 151
6.5 Discussions 151

6.6 Validity Threats and Limitations 157
6.7 Related Work 158
6.8 Conclusion 160

7 Conclusions and Future Work 161

7.1 Thesis Contributions 163
7.2 Future Work 165
7.2.1 Preprocessor-Based Program Extraction 165
7.2.2 Evolution Visualization on a Large Scale 165
7.2.3 Evolution Analysis of Software Architecture 166
7.2.4 Evolution Simulation Based on Known SOC Models 167

A A List of Open Source Projects 169

B Distributions of Software Changes 175

List of Tables

1.1 Laws of software evolution as summarized by Lehman [LRW+97] 2

2.1 Uses of comprehension pipelines on software systems 30

2.2 Open source systems extracted using the CX extractors 34

3.1 Experimental conditions for linkage resolution 64

3.2 Linkage resolution under condition C1 (PostgreSQL) 68

3.3 Individual linking steps of LDH and HEU (PostgreSQL) 69

3.4 Anomaly ratios at varying levels of granularity (PostgreSQL) 70

4.1 Setting for studying punctuated software evolution 87

4.2 Ratio of changed files in PostgreSQL 91

5.1 Software systems chosen for the clustering experimentation 106

5.2 Relative stability scores obtained using MoJo 116

5.3 HML-based stability scores obtained using MoJo 116

5.4 Relative authoritativeness scores obtained using MoJo 120

5.5 A summary of ordinal evaluation of clustering algorithms 125

6.1 Scaling exponents for distributions of software changes 146

6.2 Hurst exponents from R/S analysis of daily time series 150

6.3 Analogy between the sandpile model and software system 153

List of Figures

1.1 Thesis Overview 7

2.1 BFX fact schema 20
2.2 LDX as a substitute for LD 22
2.3 CTSX built on Ctags and Cscope 23
2.4 Program comprehension pipelines built on the CX extractors 27
2.5 Performance comparison of the CX extractors 37

3.1 A sample schema of LPM 47
3.2 Linking heuristic: File-File 52
3.3 Linking heuristic: File-File Closure 53
3.4 Linking heuristic: Same Subsystem 53
3.5 Linking heuristic: Nearest Super Subsystem 54
3.6 A partial build dependency graph of PostgreSQL 60
3.7 Quantitative comparison of cross-references in PostgreSQL 72

4.1 Spectrograph model based on matrix 85
4.2 Evolution spectrographs of OpenSSH 89
4.3 Evolution spectrographs of PostgreSQL 92
4.4 Evolution spectrographs of Linux 95

5.1 Comparison of clustering algorithms in the context of software evolution . 107
5.2 Intra-sequence clustering comparison 113
5.3 Stability comparison wrt PostgreSQL 115
5.4 Inter-sequence clustering comparison 118
5.5 Authoritativeness comparison wrt PostgreSQL 119
5.6 Distribution comparison of clustering algorithms wrt PostgreSQL 122
5.7 NED-based distribution comparison wrt PostgreSQL 124

6.1 A delta graph between two adjacent versions 141
6.2 Tail cumulative distribution of change sizes for GCC 144
6.3 R/S analysis of daily time series for GCC 149

B.1 Tail CDF of changes recovered from the CVS repository 176
B.2 Tail CDF of daily system structural changes 177

Publications

The work presented in this thesis has appeared peer-reviewed articles listed below:

1. Fractal Nature of Software Evolution and SOC Dynamics. Jingwei Wu, Richard C. Holt and Ahmed E. Hassan. Draft.
2. An Extractor Suite for C and C++: Choosing the Right Tool for the Job. Jingwei Wu and Richard C. Holt. Draft.
3. Comparison of Clustering Algorithms in the Context of Software Evolution. Jingwei Wu, Ahmed E. Hassan and Richard C. Holt. Proceedings of International Conference on Software Maintenance (ICSM). Budapest, Hungary, September 2005.
4. Visualizing Historical Data Using Spectrographs. Ahmed E. Hassan, Jingwei Wu and Richard C. Holt. Proceedings of International Symposium on Software Metrics (METRICS). Como, Italy, September 2005.
5. Exploring Software Evolution Using Spectrographs. Jingwei Wu, Richard C. Holt and Ahmed E. Hassan. Proceedings of IEEE Working Conference on Reverse Engineering (WCRE). Delft, Netherlands, November 2004.
6. Evolution Spectrographs: Visualizing Punctuated Change in Software Evolution. Jingwei Wu, Claus W. Spitzer, Ahmed E. Hassan and Richard C. Holt. Proceedings of International Workshop on Principles of Software Evolution (IWPSE). Kyoto, Japan, September 2004.
7. Resolving Linkage Anomalies in Extracted Software System Models. Jingwei Wu and Richard C. Holt. Proceedings of International Workshop on Program Comprehension (IWPC). Bari, Italy, June 2004.
8. Linker-Based Program Extraction and Its Uses in Studying Software Evolution. Jingwei Wu and Richard C. Holt. Proceedings of International Workshop on Unanticipated Software Evolution (FUSE). Barcelona, Spain, March 2004.

Chapter 1

Introduction

As open source software gains popularity, researchers have great opportunities to study and explore large numbers of open source projects to gain an enhanced understanding of software evolution. In this research, we develop techniques and approaches to support software evolutionary data collection and investigation on a large scale. We present several new ways for understanding the evolution of open source software systems by making use of collected evolutionary data.

1.1 Prior Research

Lehman *et al.* formulated and refined eight laws of software evolution to model the dynamical behavior (change or growth) of software systems as these systems are maintained and enhanced over time [LB85, Leh97]. As summarized in Table 1.1, the eight laws are *Continuing Change*, *Increasing Complexity*, *Self Regulation*, *Conservation of Organizational Stability*, *Conservation of Familiarity*, *Continuing Growth*, *Declining Quality*, and *Feedback System* [Leh97]. Lehman's laws represent the best known body of work on understanding

No.	Brief Name	Law Description
I	Continuing Change	An E-type system must be continually adapted else it becomes progressively less satisfactory.
II	Increasing Complexity	As an E-type system evolves its complexity increases unless work is done to maintain or reduce it.
III	Self Regulation	Global E-type system evolution processes are self-regulating.
IV	Conservation of Organizational Stability	The average effective global activity rate in an evolving E-type system is invariant over system lifetime.
V	Conservation of Familiarity	During the active life of an E-type system, the content of successive releases is statistically invariant.
VI	Continuing Growth	The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime.
VII	Declining Quality	The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environmental changes.
VIII	Feedback System	E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to be successfully modified or improved.

Table 1.1: Laws of software evolution as summarized by Lehman [LRW+97]
(An E-type system solves a problem or addresses an application in the real world.)

the evolution of large, long-lived software systems.

The laws of software evolution were initially grounded on observing how large software systems were developed and maintained in corporate environments using conventional management techniques and processes [LB85]. Attention was mainly directed to phenomena related to growth, continual adaptation, user satisfaction, global activity rate and feedback. The advent of open source software projects provides new opportunities for studying software evolution over large numbers of systems. A large amount of effort have been devoted to studying open source software evolution, and a variety of interesting and sometimes even conflicting findings have been reported, which we review briefly in the following section.

1.1.1 Open Source Software Evolution

Open source software projects rely on a development approach different from the way most closed source industrial software systems are created [Ray99]. This approach promotes free access to the source code and collaborative development which is commonly supported by a decentralized Internet-based developer and user community. Recent empirical studies of open source software evolution have reported on evidence for either validation or disproof of Lehman's laws. Many interesting evolutionary phenomena not characterized by Lehman's laws are also discovered.

Godfrey and Tu observed the super-linear growth of the Linux kernel from 1994 to 1999 [GT00]. The growth of Linux is unusual because it is different from the most commonly observed linear and inverse-square growth models [Tur02]. Three factors have been found to contribute to such a growth: (a) much of the code is device drivers; (b) much of the code is orthogonal and intended for different platforms; and (c) the code base is open to interested developers who can make their own contributions. Their findings are further confirmed in a recent study conducted by Robles *et al.* [RAGBH05]. Linux grew at a super-linear rate from 100,000 to 4,500,000 lines of code during the period between 1994 and 2004. Similar findings are reported by O'Mahony on the Debian GNU/Linux distribution from release 0.01 in 1993 to 3.0 in late 2002. This study shows that the Debian GNU/Linux distribution size has grown at a super-linear rate [O'M06]. These findings cause speculation that open source systems may evolve in a way different from closed source industrial systems.

Not all open source projects can grow at a super linear rate. Open source projects from different domains are found to grow at different rates – some super-linear, some linear and some sub-linear, but the linear growth appears to be the dominating trend [RAGBH05].

Schach and Offutt examined the Linux kernel in depth. They found that the number of

common couplings[1] in Linux exhibited an exponential growth, thus suggesting an increased complexity [SO02]. They suggested that Linux developers should be aware of this alarming growth trend and design solutions to reduce the number of common couplings. Otherwise, the Linux kernel may become too expensive to maintain in the future.

Bauer and Pizka summarized a number of principles and practices which contribute to the success of open source projects [BP03]. They have hands-on experience with some high profile products such as GCC and Linux and many small systems. According to them, the process and organization scale as the project increases in size; and the system architecture is hardly pre-planned but evolves with the development process and requirements. For example, the Mozilla application was split into isolated parts which are either independently maintainable modules or completely new products. Nakakoji *et al.* found that open source projects followed different patterns of split and merge across releases, and they proposed to classify open source projects into three types: Exploration-Oriented, Utility-Oriented and Service-Oriented [NYN+02]. Based on such a classification, the authors provided guidance on the creation and maintenance of sustainable OSS development and communities.

Mockus, Fielding and Herbsleb reported that a core team of 10 to 15 developers could enable a high rate of growth [MFH02]. Such core teams are important to anchor a broader community of users and developers and to ensure the long term evolution of an open source project. Madey *et al.* studied how developers participated in multiple open source projects [MFT02]. They observed that developers collaborated through social networks interlinking different projects. Such networks allow the interlinked projects to share source code.

These early studies of open source systems have enhanced the understanding of software evolution by taking into consideration many different types of entities including the system, architecture, development process, and developer organization. Although many interesting

[1] Common coupling refer to the shared use of a global variable between two different source files.

evolutionary phenomena or patterns are observed, the mechanism underlying them, *i.e.*, the evolution dynamics, is still not well understood. A simple, unified explanation for diverse evolutionary phenomena observed in open source software systems is needed.

1.2 Problem Definition

We now describe two main research problems under our investigation. Their solutions form the main body of the thesis.

Change is the essential nature of software evolution. In response to the changing environment and requirements, a software system has to be changed to add new functionality, to reduce growing complexity, and to maintain the quality of service. Throughout the lifetime of a software system, changes can be made for different purposes, which are related to features, refactorings, defects, cleanup and etc. From an external perspective, changes are planned by an organizational and management structure to accommodate large unexpected requirements. From an internal perspective, developers collaborate with one another to perform assigned or self-selected tasks in various parts of the system. What evolutionary patterns can one observe from either an external or internal perspective? What underlying mechanism constrains evolutionary changes and how? These questions have been only partially answered by prior work. Lehman's seminal work provides most insights into software evolution through observing how large, long-lived software systems grow over time. In this thesis we describe and explain the evolution of open source systems from both the external and internal perspectives.

Prior studies of software evolution are commonly limited to about 10-20 versions (data points) [Leh97, GJKT97, Per02] or measure basic attributes such as the release date and the system size [GT00, RAGBH05]. Complex data such as structural dependency graphs

has not been adequately used in studying software evolution. Recovering historical changes to the system structure or architecture over an extended period of time can provide valuable information for understanding how a software system is maintained [CKN+03, ZG03]. To substantiate a large scale investigation of software structural evolution over many systems and over many versions, we need automated, efficient and scalable techniques.

The problems addressed in this thesis can be summarized as follows:

1. Develop a cost-effective solution to collect structural information over system lifetime.
2. Develop an understanding of open source software evolution dynamics from both the external and internal perspectives.

A solution to the first problem helps us obtain large quantities of historical data, upon which we can conduct large scale empirical analyses of the evolution of many open source systems and seek ways for approaching the seconde problem.

1.3 Thesis Organization

Throughout this thesis we build an understanding of software evolution by analyzing large quantities of historical information collected from open source software systems. This thesis has three main parts as shown in Figure 1.1. The chapters contained by each part are also included in Figure 1.1.

- **Part I – Collect Data**

  We present approaches and techniques for extracting software system models over a software system's lifetime in a timely and cost-effective manner. We focus on program extraction (Chapter 2) and program model linking (Chapter 3) respectively.

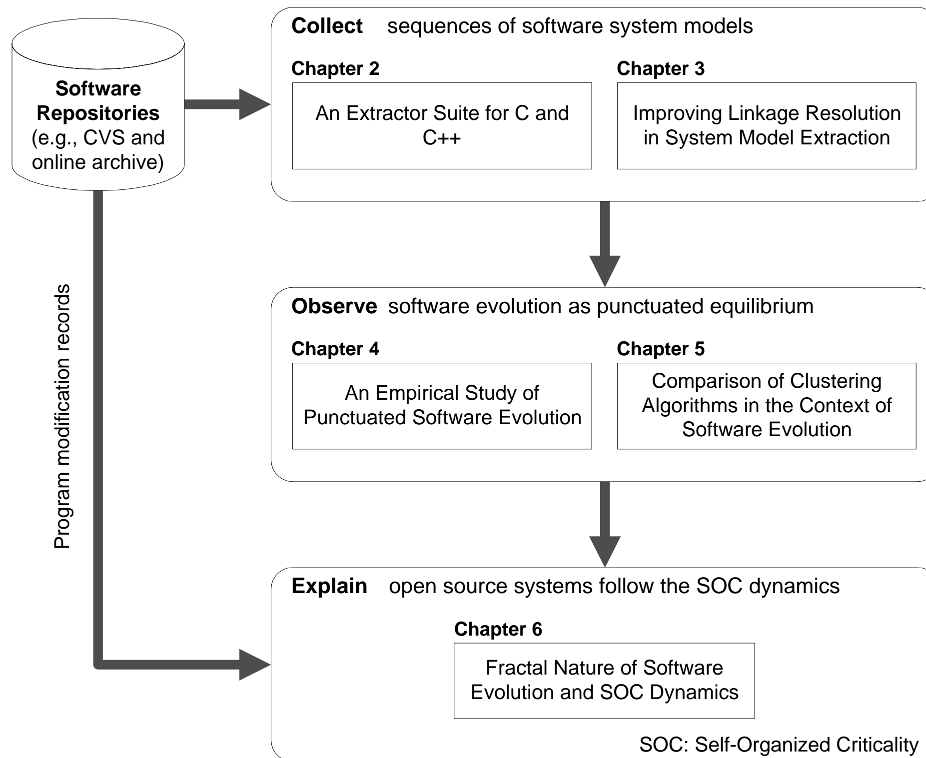


Figure 1.1: Thesis Overview

- **Part II – Observe Phenomena**

  We present approaches and techniques for examining how the structure of a software system is changed at two levels of abstraction: the implementation level (Chapter 4) and the design level (Chapter 5). Our objective is to build an empirical understanding of software evolution as punctuated equilibrium. We present evidence that a software system evolves through long periods of small changes interrupted occasionally by large avalanche changes. This part helps us to gain an understanding of software evolution from an external perspective.

- **Part III – Explain Evolution Dynamics**

We present a new perspective on evolution dynamics, which is based on Self-Organized Criticality (SOC) [BTW87]. From this perspective, a system follows an intrinsic complex process to respond to external events or forces (Chapter 6). The SOC process has special statistical features which are related to power laws. This part offers a view of software evolution from an internal perspective.

Each of these three parts of this thesis is covered by one or two chapters. Related work for each part will be examined in the corresponding chapters.

1.4 Thesis Overview

We now provide an overview of the remaining chapters.

Chapter 2: An Extractor Suite for C and C++

In Chapter 2 we describe a suite of program extractors for C and C++ which are developed by means of adopting and extending current extractors and/or tools. Each extractor from this suite is targeted at an individual step in the build process of C and C++ programs such as compilation and linking.

We assess the benefits of this suite by applying it to support two major applications: (1) creating program comprehension pipelines to support various understanding tasks, and (2) building an open source software evolution database to support empirical research on software evolution. We discuss the lessons learned in detail.

Chapter 3: Improving Linkage Resolution in System Model Extraction

Separating compilation and linking is critical to the development of modern C/C++ software systems up to millions of lines of code. Similarly, program extraction is also separated

from program model linking to deal with large programs. An extractor is first applied to extract program models from individual files and then a linker is used to create a software system model by combining separate program models. This separation has diverted a significant amount of effort to developing new extraction techniques but left program model linking unattended in reverse engineering.

Linking errors (unexpected or missing dependencies) can be introduced to the resulting system model even if individual files can be extracted correctly. Even worse, linking errors can propagate to high level models and yield an increasingly larger impact on downstream software analyses such as software clustering and reflexion modeling.

In Chapter 3, we describe four different methods for linking individual program models and discuss their benefits and limitations. We conduct a comparison of these methods on an open source system, PostgreSQL [Pos03]. The obtained results show that inappropriate linking can lead to a relatively large percentage of dependency anomalies at higher levels of granularity. While, a small set of linking heuristics independent of system configuration are found effective for reducing the occurrences of dependency anomaly. These heuristics combined with lexical program extractors are particularly useful for creating large software system models over hundreds of versions.

Chapter 4: An Empirical Study of Punctuated Software Evolution

Theories of biological evolution provide rich sources of ideas from which software evolution research can benefit. Motivated by the theory of *Punctuated Equilibrium* [EG72] as well as previous research on discontinuous software evolution [Leh97, Law82, AP01], we propose to view software evolution as punctuated equilibrium. A software system evolves through an alternation between long periods of small incremental changes and short periods of large avalanche changes.

We provide an interpretation of punctuated equilibrium in the context of software evolution. We conduct an empirical study on three open source systems (OpenSSH, PostgreSQL, and Linux) to examine structural changes over system lifetime. A visualization technique called the Evolution Spectrograph is used to highlight major change events. The observed events are then correlated manually with design changes documented in various sources (*e.g.*, release notes). The results we obtained show that the three examined systems exhibit strong characteristics of punctuation during their evolution.

Chapter 5: Comparison of Clustering Algorithms in the Context of Software Evolution

To support software understanding and maintenance tasks, various automated clustering algorithms have been developed to partition a software system into meaningful subsystems (or clusters). However, it is unknown whether these algorithms produce similar meaningful clusterings for similar versions of a software system under continual change and growth. If we find an algorithm with such capability, we can apply it to automate design recovery over system lifetime. The recovered design data (clusterings) can provide new opportunities for examining software evolution at the system design level. To pursue such research, we need to compare automated clustering algorithms in the context of software evolution.

In Chapter 5 we evaluate six automated software clustering algorithms. Each of these algorithms is applied to subsequent versions from five open source systems. The obtained clusterings are compared based on three criteria respectively: stability (Does the clustering change only modestly as the system undergoes modest updating?), authoritativeness (Does the clustering reasonably approximate the structure an authority provides?), and cluster distribution extremity (Does the clustering avoid huge clusters and many small clusters?).

The six studied algorithms exhibit distinct characteristics. For example, the clustering

from the most stable algorithm bears little similarity to the as-implemented structure of the system; and the clustering from the least stable algorithm has the best cluster distribution. The obtained results indicate that current automated clustering algorithms need significant improvement to provide continual support for maintaining large software projects. They are also not suitable for automating a large scale empirical analysis of software evolution at the design level.

Chapter 6: Fractal Nature of Software Evolution and SOC Dynamics

We study software evolution dynamics by adopting the theory of *Self-Organized Criticality* (SOC) [BTW88]. The SOC dynamics is characterized by spatial and temporal fluctuations which exhibit fractal properties measured as power laws. Our previous empirical observation of punctuated evolution in software systems leads us to question whether software systems follow the SOC dynamics during their evolution.

In Chapter 6 we examine the change history of eleven open source software systems and show empirical evidence for the presence of fractal structures in the dynamical behavior of software systems. Specifically, we report two power law related findings throughout system lifetime: (1) the probability density of change occurrences decreases as a power function of the change size; and (2) the time series of change exhibits long range correlations with power law behavior. The existence of such spatial and temporal power laws strongly suggests that the SOC dynamics may provide a useful conceptual framework for understanding software evolution.

1.5 Contributions

The contributions of this thesis are summarized as follows.

- The proposal and development of a suite of program extractors for C and C++. By covering individual steps in the program build process, such a suite supports program extraction in a systematic manner and allows for tradeoff between accuracy, efficiency and robustness to varying degrees.
  - A number of program comprehension pipelines are built on this suite to support various downstream software analysis.
  - An open source software evolution database called EvolDB is created using this suite to support empirical research on software evolution. EvolDB contains tens of thousands of system models.
- The development of program model linking methods to aid the extraction of software system models. A small set of heuristics are proposed to link separate program models into a combined system model when system configuration information is difficult to obtain. They provide convenient support for the mass-production of system models.
- The proposal of viewing software evolution as punctuated equilibrium. In this view, a software system undergoes long periods of small changes separated by large avalanche changes. The evolution spectrograph is used to visualize structural changes through system lifetime, providing a visual display of punctuated change.
- A theoretical explanation of open source software evolution dynamics based on *Self-Organized Criticality* (SOC). Such an explanation is supported by our observation of two fractal related phenomena in the change history of open source software systems. The observed phenomena are (1) the power law distribution of changes and (2) long range correlations in the time series of change.

- The proposed framework for evaluating software clustering algorithms based on three criteria: stability, authoritativeness and extremity. Within this framework, a number of representative automated software clustering algorithms are compared using data stored in EvolDB. Such a comparison demonstrates the usefulness of large quantities of historical data in advancing software clustering research.

Chapter 2

An Extractor Suite for C and C++

This chapter describes a suite of program extractors which are developed by adopting and extending current techniques and tools. This suite is called CX since it is mainly targeted at extracting facts from C and C++ programs. It is currently composed of four extractors: CPPX, BFX, LDX and CTSX. The main goal of creating CX is to form a convenient set of program extractors which complement each other and work in a systematic manner. The benefits of this extractor suite will be discussed in terms of two practical applications: (1) creating program comprehension pipelines to support various understanding tasks, and (2) building an open source software evolution database to support empirical research on software evolution.

2.1 Introduction

Program extraction is important to program comprehension and maintenance tasks [FSG04, MN96]. For example, an architect may need to monitor changes made to a software system to identify changes violating the design constraints of the system. In this case, a program

extractor is needed to collect structural information to support continual maintenance. In empirical research on software evolution, a robust and efficient extractor is often required to collect structural artifacts from a large number of versions, sometime up to several hundred versions. A researcher may also be interested in deriving a domain reference architecture by studying several systems from the same domain (*e.g.*, the compiler domain and the web browser domain). In these various examples, an extractor needs to be chosen by making an appropriate tradeoff among requirements on accuracy, efficiency and robustness. In an empirical study of software evolution, a researcher is not likely to use an accurate extractor which spends months in extracting structural artifacts from many historical versions of a long-lived system (*e.g.*, Linux [Lin04]). A robust and efficient extractor, though it may be less accurate, is more likely to prevail.

No single extraction technique can meet the highest standards on accuracy, efficiency and robustness and in the meanwhile support as many tasks as possible [SHE02]. Instead, a very small set of complementary extractors should be developed to support a wide range of tasks and meet extraction requirements to varying degrees. In this chapter, we describe our effort in developing a suite of program extractors for the C and C++ programming language. The main design goals of this suite are as follows.

- It should be simple and convenient to use.
- It should scale up to handle large programs.
- It should support diverse program analysis tasks.
- It should cover the entire build process of software.
- It should efficiently handle large numbers of versions.

CPPX [CPP02] is the first extractor we added to the CX suite. CPPX is a C/C++ source code extractor based on the front end of GCC [GCC02]. It was developed by the

SWAG group at the University of Waterloo. CPPX outputs detailed information of a program at the abstract syntax graph level. However, CPPX is neither efficient nor robust in handling very large programs over many versions. This motivated us to develop several lightweight extractors to complement it. These new extractors and CPPX form a suite that provides a simple and cost-effective solution for a wide variety of program analysis tasks (see Section 2.3).

This chapter is organized as follows. Section 2.2 describes each extractor from the CX suite and their pros and cons. Section 2.3 describes two major applications of the CX suite. Lessons learned from each application are also summarized. Section 2.4 further compares four CX extractors and discusses their role in supporting systematic extraction of C/C++ programs. Section 2.5 considers related work. Section 2.6 draws the conclusion.

2.2 The CX Suite

The CX suite consists of four program extractors which are created by means of adapting free open source tools, including GNU Compiler Collection (GCC) [GCC02], GNU Binutils (binary utilities) [BUM02], Ctags [Cta04] and Cscope [Csc04]. These extractors are briefly summarized below.

- **CPPX** is a C/C++ source code extractor based on the GCC frontend. It relies on the preprocessing, parsing, and semantic analysis of GNU g++ and can produce program facts as detailed as abstract syntax graph (ASG).

- **BFX** is a binary code extractor built on the Binary File Descriptor (BFD) library. It parses binary code to locate definitions of functions and variables and outputs symbol references to these definitions.

- **LDX** is a binary code extractor based on the GNU code linker LD. It reuses BFX to process individual binary files and then resorts to the real code linker in the resolution of cross-references among different binary files under a specific system configuration.
- **CTSX** is an efficient and robust source code extractor built upon Ctags and Cscope. The Ctags is invoked to locate source program entities and Cscope is used to extract references to source program entities.

We now describe each extractor in the CX suite and their pros and cons in detail.

2.2.1 CPPX

CPPX is a general-purpose parser and fact extractor for C and C++ programs. It relies on the preprocessing, parsing, and semantic analysis of GNU g++, and produces an abstract syntax graph in accordance with the Datrix model [Bel01]. The produced fact base is in TA [Hol02], GXL [GXL02], or VCG [San95] format.

Abstractly speaking, CPPX output is an E/R graph, which is essentially the abstract syntax graph of the source program being extracted. The vertices represent the program's templates, classes, methods, compound statements, and expressions down to the lowest level of constants and variable references. The graph edges represent syntactic relationships as well as semantic facts linking identifiers to their declarations, function calls to their targets, objects to their types, and most things to their enclosing scopes. From the CPPX output graph it is (almost) possible to reconstruct the original program [LHM03].

CPPX is suitable for use in architectural recovery, data flow analysis, pointer analysis, program slicing, query techniques, source code visualization, object recovery, refactoring, restructuring, re-modularization, and the like. It has been used in both industrial software development environments and academic software engineering research. However, CPPX

has several problems. For example, CPPX may take an unusually long time (up to months) to extract facts from large software systems (up to several million lines of code); and the abstract syntax graph, which is overly detailed for many downstream analysis, often needs lengthy transformations to filter out large quantities of unwanted data such as compound expressions and statements. These problems make it necessary to develop more efficient and simplified program extractors. As a result, we have developed three lightweight program extractors, each of which is detailed in the following.

2.2.2 BFX

BFX (Binary File Extractor) is built on the Binary File Description (BFD) library which is shipped with the GNU Binutils toolkit [BUM02]. Unlike CPPX which deals with source code, BFX extracts facts from binary code (machine code). It can process object modules (`.o`), archives (`.a`), dynamic libraries (`.so`) and executables (`.exe`). The output is in TA or GXL format and conforms to the fact schema shown in Figure 2.1.

The schema in Figure 2.1 shows that the BFX output has three levels of granularity: the level of object modules, the level of functions and variables and the level of name references. The `cObjectFile` class represents files ending with `.o`, `.so` and `.exe`. The `cArchiveFile` represents archive files (`.a`). The `cFunction` and `cObject` represent functions and variables respectively. The `cExternSymbol` represents unique string names. There are four relations: a structural relation `contain` and three reference relations `cRefersTo`, `cRefersToExtern` and `cResolvedByExtern`. The `contain` relation for any binary module always forms a tree with a universal root of type `cScopeGlb`. The `cRefersTo` relation refers to resolved references to symbols defined within the same binary module. The `cResolvedByExtern` relation denotes references to externally defined symbols and `cRefersToExtern` means that

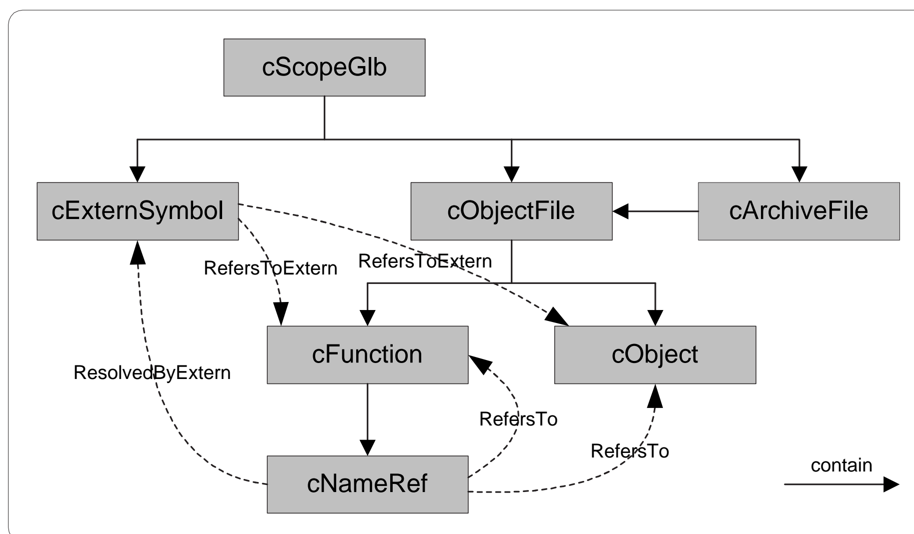


Figure 2.1: BFX fact schema

a definition can be looked up globally by searching for its name[1], *i.e.*, `cExternSymbol`.

In the above schema, `cNameRef` has an attribute called `kind`, which can be assigned a value of `F` or `V`. The `F` value means a function call and the `V` value means a variable use.

Pros

BFX is built on the BFD library. Technically speaking, BFX is equivalent to the binary code dump tool *objdump* [BUM02]. BFX is independent of source code compilation. Therefore, it is able to extract programs written in programming languages other than C and C++, which include, for example, Fortran and Pascal. The BFX output is more than an order of magnitude smaller than the CPPX output since it extracts only function calls and variable uses. In terms of speed, BFX normally operates in a matter of seconds or at most minutes to extract large systems with thousands of object units. BFX's performance will

[1]For C++ programs, an external definition has a mangled name.

be further discussed in Section 2.4.

Cons

BFX is used only after source files are compiled into object units. This results in the loss of a significant amount of structural information related to various programming constructs such as abstract data types (*e.g.*, `union` data types) and macros. The extracted function calls and variable uses reflect only what object units contain and what external symbols are referenced. There exists a gap between this kind of structural information and what the programmer sees in the source code. For example, a function-like macro, which is often treated and used like a real function by the programmer, can not be extracted using BFX since it is expanded in the step of preprocessing.

2.2.3 LDX

LDX (Linker Based Extractor) is built on the GNU code linker LD [BUM02]. LDX performs both code linking (including symbol name resolution across the boundary of object modules) and fact extraction. Its output includes everything produced by BFX as well as build dependencies between object modules, archive files, dynamic libraries and executables. The output is in TA or GXL format. As shown in Figure 2.2, LDX operates as a full substitute for LD during the extraction of a program.

Compared to BFX, LDX has two distinct features. First, it relies on the actual configuration of the program and the internal linking logic of GNU LD to resolve cross-references among separately extracted object units. Therefore, the extraction and linking of program facts are correctly and simultaneously carried out as the target system is being built. Second, LDX captures build dependencies among object units as perceived by the code linker during link time. For example, a simple hello world program (*e.g.*, `Hello.exe`) compiled on

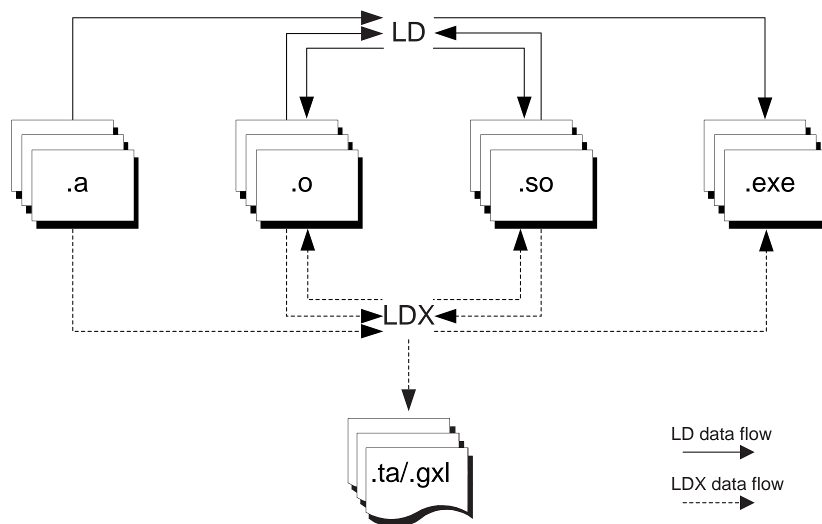


Figure 2.2: LDX as a substitute for LD

a Linux machine normally depends on an object module (*e.g.*, `Hello.o`) and a particular version of the C dynamic library (*e.g.*, `/lib/libc.so.6`).

Pros

Like BFX, LDX is multi-lingual and operates at a very fast speed. It adds only negligible overhead to the build process of a target system. Being a link time extractor, LDX utilizes the system configuration information (*i.e.*, build dependencies among object modules) and LD's symbol resolution functionality to derive cross-reference facts correctly. LDX does not introduce any erroneous cross-references.

Cons

LDX has similar cons BFX has. In addition, LDX causes slightly more interference to the build process of a software system than BFX since it needs to be substituted for LD.

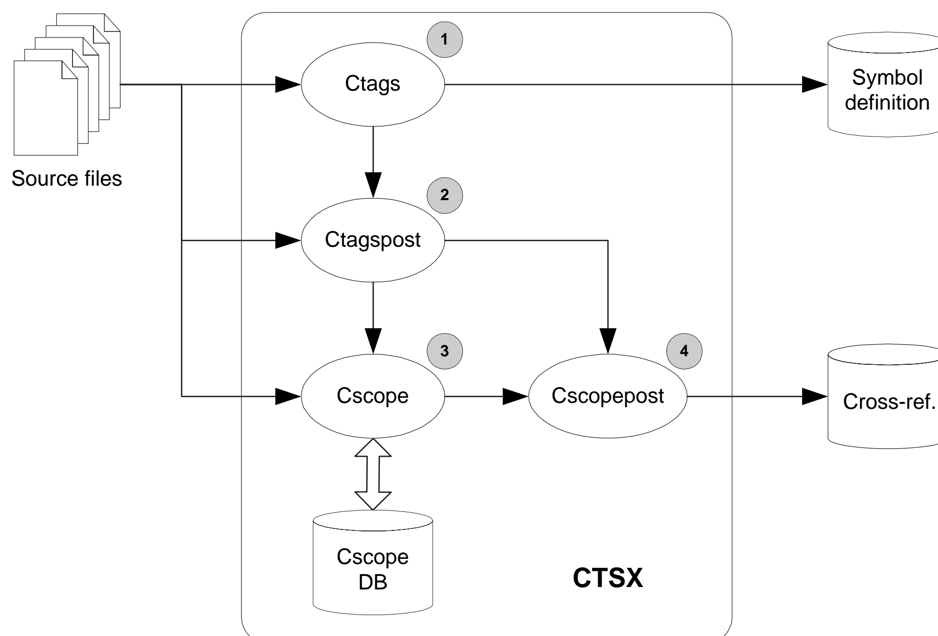


Figure 2.3: CTSX built on Ctags and Cscope

2.2.4 CTSX

The three extractors described above require a successful build of the target system. The extraction can run into serious trouble when large numbers of versions need to be extracted. For a long lived system, it is common that many versions can not be compiled successfully on a specific platform. A robust extractor is required to tolerate erroneous and incomplete source code as well as system configuration problems. In addition, the extractor needs to be efficient in handling large numbers of versions. For example, the Linux kernel has more than 550 versions publicly posted on its official archive Web site [Lin04] until July 2005. It is a daunting task to use CPPX, BFX or LDX to extract Linux versions since considerable manual effort is needed to deal with various configuration and compilation difficulties over the lifetime of Linux (about 12 years).

We developed a lightweight C and C++ source code extractor to support the extraction of a large long-lived software system over hundreds of versions. This extractor is based on Ctags [Cta04] and Cscope [Csc04]. For this reason it is called CTSX in which T stands for C**t**ags and S for C**s**cope.

Ctags is a tagging tool used by source code editors to parse the source code being edited. Using Ctags, editors can provide rudimentary support for code highlighting and searching. Cscope is a cross-referencing tool for browsing the source code. It allows the user to search for source code entities (definitions and regular expression patterns) as well as references to these entities in an interactive mode. Ctags and Cscope are both efficient and robust. They scale up to deal with more than 20 million of lines of code [Csc04].

The implementation of CTSX is illustrated in Figure 2.3. CTSX has four components: Ctags, Ctagspost, Cscope and Cscopepost. We instrumented Ctags and Cscope to parse new command line options and to read/write facts in a proper format such as TA and CSV (Comma Separated Values). Ctagspost and Cscopepost were written in Perl and they are aimed at reducing errors and adding extra attributes. The execution order of these tools is indicated by numbers. CTSX takes in a list of source files and produces two main text files, which contain symbol definitions and cross-references respectively. The main functionality of each component is detailed as follows.

- **Ctags** extracts various program entities including functions, global variables, local variables, macros, abstract data types (`class`, `struct`, `union`, `enum` and `typedef`), enumerators (values inside an enumeration), function prototypes, namespaces, and external variable declarations. Ctags has command line options for specifying which kinds of program entity to extract. By default, CTSX instructs Ctags to extract all entities listed above.

- **Ctagspost** post-processes Ctags output to add extra information in three ways: (1)

extract function parameters ignored by Ctags; (2) determine modifiers (*e.g.*, `inline` and `static`) for functions or variables; and (3) calculate the effective scope of local variables and function parameters. To be simple, the effective scope of a local variable is treated to be equal to the scope of the function enclosing the variable.

Ctagspost needs to parse the source code briefly to collect extra information because additional texts associated with each program entity located by Ctags are insufficient for collecting the extra information mentioned above.

- **Cscope** parses the entire source code to build an initial cross-reference database. It then reads program entities from Ctagspost output and retrieves static references to those entities. Depending on the command line options the user specifies for CTSX, Cscope can retrieve different kinds of cross-reference. By default, it outputs references to functions, global variables, macros, and data types (including `typedef`s).

  Every name reference produced by Cscope is assigned a `kind` attribute to indicate its type. Cscope itself can determine which reference is a function call. However, the type of other references is determined by using regular expressions to match relevant code syntax. For example, a type reference can be determined if the type name appears as part of a declaration or a type cast. The references to variables and macros are not distinguished in the current implementation.

- **Cscopepost** filters out references to static functions, static variables, local variables and function parameters through the use of the scoping and accessibility information produced by Ctagspost. For example, a reference to a static function within the same file is filtered out as a non cross-reference. A reference to a global variable is filtered out as an erroneous reference if it falls in the scope of a local variable which has the same name as the global variable. The Cscopepost output contains only references

that cross the actual boundary of source files.

Pros

Compared to writing an extractor from scratch, the reuse of Ctags and Cscope significantly speeds up the development of CTSX. Only a few days were spent in instrumenting Ctags and Cscope and writing the postprocessing utilities in Perl scripts.

CTSX is a useful program extractor when (1) the target system is extremely large and time consuming to build, (2) the system cannot be built due to configuration or compilation errors, and (3) the correctness of extracted facts is not of critical concern to downstream software analysis. These characteristics make CTSX suitable for extracting program facts of reasonable quality from the evolution history of a long-lived software system in a timely and cost-effective manner. The benefits of CTSX will be demonstrated in section 2.3.2.

Cons

As a lightweight program extractor, CTSX is more error prone than CPPX, BFX and LDX. This is mainly caused by three factors. First, Ctags is reliant on fault tolerant parsing and it thus may result in the missing of program entities or the recognition of wrong entities. Second, Cscope has no knowledge of the typing of symbol references except function calls. The lexical analysis based on regular expressions can only alleviate this typing problem. Third, Cscope does not differentiate references to local and global program entities. To reduce the undesirable impacts of these factors, Ctagspost and Cscopepost are added to search for more semantic clues. However, without complete semantic analysis, it should not be expected that CTSX or any lightweight parsing techniques (commonly based on regular expressions [MN96] and island grammars [Moo01]) can produce results as accurate as those produced by the extractors based on full parsing and semantic analysis (*e.g.*, CPPX).

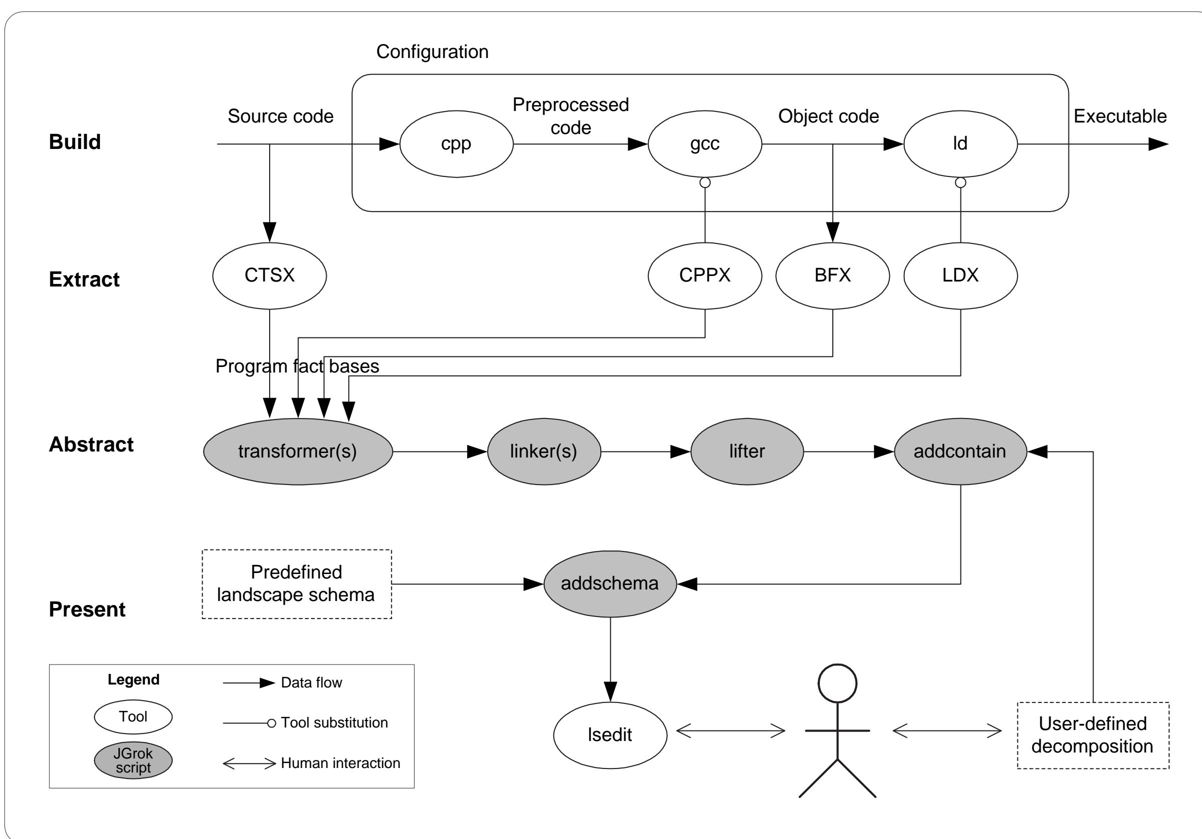


Figure 2.4: Program comprehension pipelines built on the CX extractors

2.3 Applications

This section describes two main applications of the CX suite to demonstrate its benefits in practice: (1) creating various program comprehension pipelines to aid software developers and researchers; and (2) building an open source software database to support empirical research on software evolution. These applications involve a number of open source software systems. Each involved system is briefly described in Appendix A.

2.3.1 Creating Comprehension Pipelines

A program comprehension pipeline commonly consists of three main steps: *extract*, *abstract* and *present*. The SwagKit uses such a pipeline to manipulate CPPX facts [Swa02]. As new extractors (BFX, LDX and CTSX) are developed, we have extended the old SwagKit by implementing new comprehension pipelines to support more diverse software analysis tasks cost-effectively, for example, the recovery of reference architecture for an application domain [GG05].

Figure 2.4 illustrates the main components shared by various comprehension pipelines currently supported by the extended SwagKit. Each extractor from the CX suite serves as the starting point of a specific comprehension pipeline. The four main steps are as follows:

- **Build** is an extra yet important step in a program comprehension pipeline. The build process of a C/C++ software system normally consists of configuration, compilation and linking. For extractors which are created by adapting build related tools such as the compiler and linker, the build process provides a simple vehicle for carrying out program extraction. However, build is not needed for extractors which ignore system configuration information and perform their own parsing and semantic analysis from scratch.

- **Extract** refers to the extraction of program related facts by using appropriate tools. Broadly speaking, the extracted facts can be as abstract as software architecture or as concrete as cross-references among various program entities (*e.g.*, functions). The latter is what a program extractor normally produces.

  The CTSX extractor can be used independent of system configuration. By contrast, the other three extractors must be applied after the system is configured. CPPX and LDX need to be embedded into the build process through tool substitution. Program

facts are extracted as the system is being built. BFX can be embedded into the build process or applied directly on all object modules after a successful compilation.

- **Abstract** is the step of manipulating program facts through transformation, linking and lifting. Facts produced by different extractors are transformed into an appropriate form with unwanted program entities and relationships removed and new program entities and relationships created. A sequence of *transformers* is applied until a desired form is achieved. The transformed program facts for individual files or units are linked to form a graph model representing a part of the system or the whole system. The *linker* resolves references across the boundary of files. The *lifter* abstracts lower level cross-references into higher level dependencies. The *addcontain* script imposes a subsystem decomposition hierarchy on the flat system model. The resulting model can be further lifted if it is too large to handle in a subsequent step.

- **Present** is the last step. The *addschema* adds a standard schema to the abstracted system model to produce a software landscape view which a user can explore by using the visualization tool *lsedit* developed by the SWAG group[Swa02]. The user can examine relationships among various program entities at different levels of granularity.

After program facts are transformed, all the pipelines share the remaining tools: *linker*s, *lifter*, *addcontain*, *addschema* and *lsedit*. These tools are implemented in JGrok, a scripting language designed for manipulating sets and relations [JGr04]. Several different linkers can be used to combine individual small program models into a large system model. Depending on the linker, the resulting system model can contain erroneous dependencies to varying degrees. How to link program models will be further discussed in Chapter 3.

Pipeline	Example for Architecture Recovery	Example for Domain Reference Architecture Recovery	Example for Software Evolution Analysis
CPPX Pipeline	Emacs (2003) InnoDB (2002) MySQL (2002) PostgreSQL (2002)		PostgreSQL (2003) [ZG03]
BFX Pipeline	DB2 UDB (2003) KSpread (2004) Mozilla (2004)	**Web Browser** (2004) [GG05]: Dillo Epiphany Flower Konqueror Lynx Mozilla Safari **Instant Messenger** (2004): CenterICQ EG-lite Gaim Kopete Miranda	KSpread (2004) OpenSSH (2004) PostgreSQL (2004) Linux kernel (2004)
LDX Pipeline	DB2 UDB (2003) Gnumeric (2004)	**First Person Shooter** (2003): Cube Quake Quake II	Gnumeric (2004) OpenSSH (2004) PostgreSQL (2004) [WH04a] Linux kernel (2004) [WH04a]
CTSX Pipeline	OpenOffice (2005)		

Table 2.1: Uses of comprehension pipelines on software systems

Common Uses

The comprehension pipelines starting with each CX extractor have been successfully used in a variety of circumstances which include industrial and academic environments as well as

graduate course teaching. They are mainly used to support three kinds of software analysis: software architecture recovery, software evolution study, and domain reference architecture recovery. Table 2.1 lists systems on which four pipelines have been applied in each kind of analysis. A brief introduction to the performed analysis is given below.

- Software architecture recovery aims at reconstructing views on the system architecture as-built. A recovered architecture is commonly represented using a hierarchical organization with components and their relationships at varying levels of granularity.
- The empirical study of software evolution is commonly conducted as a longitudinal analysis of one or more software system properties such as the system size and system structural complexity.
- Domain reference architecture recovery aims at generalizing more than one recovered system architectures from the same application domain into a standardized architecture which can be referenced for developing and understanding similar applications.

Lessons Learned

A number of lessons have been learned from observing how these comprehension pipelines were used in practice. The users mainly include graduate students from the SWAG group and students who enrolled in graduate level software engineering courses at the University of Waterloo as well as myself. During the period from 2002 to 2005, I worked as a teaching assistant to aid students in applying these pipelines on a number of open source software systems. Here are some of the lessons learned.

- Perform system extraction after a successful build. This requirement characterizes a common way of using program extractors which need to be embedded into the build

process. This has two important implications when a pipeline requires a successful build. First, build success increases the user's confidence of performing an extraction. Second, by building a system, the user can determine proper components (or features) that need to be included as the system is being configured by the user.

- The pipelines based on BFX/LDX are more convenient to use than the one based on CPPX. Two factors contribute to this observation: (1) CPPX has noticeable defects in extracting source code and often causes compilation errors; and (2) BFX/LDX is faster and produces less detailed information than CPPX. This perhaps explains why students tend to use BFX/LDX to deal with multiple versions which either belong to the same system or are from the same domain.
- CTSX is not satisfactory in conducting an accurate extraction of large C++ systems. The sheer volume of overloaded methods, local variables and type casts causes CTSX to produce a fairly large amount of inaccurate facts. In 2005, a student group enrolled in the graduate course on software architecture recovery felt uncomfortable with the accuracy of facts extracted from OpenOffice using CTSX. The group switched to the BFX-based pipeline.

In brief, the four program comprehension pipelines built on the CX suite provide useful support for analyzing large software systems in a variety of ways. Depending on the nature of software analysis, the user can choose an appropriate extractor or use different extractors in combination.

2.3.2 Building Software Evolution Database (EvolDB)

One of the main factors that impede empirical research on software evolution is the lack of appropriate tool support for collecting historical information from a long lived software

system [KS96]. Software structural artifacts such as call graph are important for software understanding and maintenance tasks [MNS95]. A sequence of structural artifacts collected from the historical versions of a system can be valuable for understanding how the system has evolved [CKN+03, ZG03].

This section describes the application of the CX suite on a number of large open source systems to build a large evolution database of software structural artifacts. This database is referred to as EvolDB. We have extracted ten open source systems over large numbers of versions (official releases or daily snapshots). These systems include GCC, KSDK, KOffice, Linux, Mozilla, OpenSSH, OpenSSL, PHP, PostgreSQL and Ruby. The benefits of building EvolDB include the following:

- The extraction of diverse software systems over a large number of versions can subject the CX suite to substantial software development in real life. The CX extractors can thus be contrasted in terms of quality attributes such as robustness and efficiency.

- The resulting evolution database is free for public use. The researchers interested in software evolution can take advantage of this database to advance empirical research on open source software evolution. Possible directions may include validating software evolution laws or explaining open source software evolution from new perspectives.

- Most important of all, EvolDB provides a large amount of historical data on which our work (described in the remaining chapters of this thesis) will depend.

Table 2.2 summarizes how many versions were extracted and how much time was spent for each target software system. The extraction was conducted on either official releases or snapshot versions checked out from the CVS source control repository. Two computers were used. CPPX, BFX and LDX were used on a Linux machine with one Intel(R) Pentium(R) IV 1.6GHz CPU and 1GB memory. CTSX was used on a Linux server with two Intel(R)

System	Period		Size (KLOC)		Versions	Extraction time (hours)		
	From	To	From	To		CPPX	BFX+LDX	CTSX
OpenSSH†	1.2pre6 (1999)	3.8p1 (2004)	20	70	60	15.1	2.2	0.11
PostgreSQL†	6.0 (1997)	7.3.4 (2004)	182	519	28	16.5	2.6	0.32
Linux kernel†	2.0 (1996)	2.5.75 (2003)	674	5140	368	-	87.2	-
	1.0 (1994)	2.6.12.3 (2005)	165	5954	581	-	-	36.32
Mozilla†	1.0 (1999)	1.7.3 (2004)	3700	4500	19	-	14.3	4.13
PostgreSQL⋄	1997-01-01	2005-01-01	182	519	97	-	13.7	-
PostgreSQL∗	1997-01-01	2005-09-09	182	556	3175	-	-	29.70
GCC∗	1997-08-12	2005-09-09	582	1559	2951	-	-	122.89
KSDK∗	1999-01-01	2004-12-31	3	263	2192	-	-	7.09
KOffice∗	1999-01-01	2004-12-31	272	962	2192	-	-	53.81
OpenSSL∗	1999-01-01	2005-09-09	164	291	2444	-	-	18.22
PHP∗	1999-04-08	2005-09-09	16	645	2347	-	-	23.58
Ruby∗	1999-01-01	2005-09-09	74	198	2444	-	-	9.01

†: official release ∗: daily snapshot version ⋄: monthly snapshot version (the first day of every month)

Table 2.2: Open source systems extracted using the CX extractors

Xeon(R) 2.2GHz CPUs and 4GB memory. Although the server has more computing power, it is approximately 40% faster than the first computer due to the running of many backend services and user tasks. A detailed performance comparison of these CX extractors will be described in section 2.4.1.

Using CPPX

We substituted CPPX for *gcc* to extract abstract syntax graphs from 28 official releases of PostgreSQL, which range from 6.0 to 7.3.4. The extraction ran into two major difficulties: (1) every version before 6.5 had configuration problems such as missing header files and

could not be built successfully, and (2) CPPX has some defects in handling very large arrays of strings and it ran out of memory for almost each of the 28 versions of PostgreSQL. We modified configuration scripts and source files to achieve a successful compilation on the Linux platform. We also used CPPX to extract a total of 60 official releases of OpenSSH ranging from 1.2pre6 to 3.8p1. Similar problems were encountered and fixed manually.

The measured extraction time given in Table 2.2 does not include the time we spent on solving problems. After a system could be built and extracted successfully, we cleaned all the generated object code and re-compiled the system from scratch in order to measure the time spent on extraction.

Using BFX and LDX

LDX and BFX were successfully used to extract four large systems (OpenSSH, PostgreSQL, Linux and Mozilla) over a large number of versions. These two extractors are able to handle very large systems at a reasonable speed. The time spent by both BFX and LDX accounts for approximately 5∼10% of the total build time of a system (see section 2.4.1).

When using BFX and LDX on Mozilla, we did not encounter any configuration problems and compilation errors. For the other three systems, the older the version, the more likely we ran into problems. All encountered problems were fixed manually.

Using CTSX

CTSX was applied to extract all the ten open source software systems. For systems which have a CVS repository, we conducted daily snapshot extraction. For OpenSSH, Linux and Mozilla, only public official releases were extracted. The extraction of 2951 daily snapshots of GCC took the longest time (122.89 hours) to finish. In general, the total time spent by CTSX on each target system is satisfactory. CTSX did not break for any version we have

extracted.

Lessons Learned

We now discuss several lessons learned from building large software evolution databases.

- For a large software system, many of its official releases may not be readily configured and compiled because the underlying platform (both hardware and software) changes drastically. This kind of build break often needs to be solved by manually modifying the configuration scripts or even the source code in order to get the system compiled. Installing outdated libraries is also required sometimes. Our experience with Linux and PostgreSQL shows that it is a daunting task to obtain a successful build for every available release. A program extractor depending on build success is not suitable for extracting a large number of versions over a long period of time.

- It is not guaranteed that CPPX can perform a successful extraction even if a software system can be compiled successfully. This is mainly caused by the internal defects of CPPX. For example, a very large array of constants can cause CPPX to break. By contrast, BFX and LDX are relatively more robust than CPPX. They can be applied to extract a version as long as the version can be compiled. CPPX transforms abstract syntax graphs which are more complicated than the format of binary code handled by BFX and LDX. This is the main reason why CPPX is less robust.

- CTSX is more satisfactory than CPPX, BFX and LDX in extracting a large software system in a robust but less accurate manner. In particular, it is capable of performing daily snapshot extraction on a long lived system within just a few hours or at most several days. CTSX is well positioned for supporting software evolution analysis on a large scale.

2.4 Discussion

2.4.1 Performance

Performance measures the speed of an extractor in extracting program facts. Performance is an important consideration when extraction is conducted on very large programs, which may have several million lines of code. In particular, if hundreds of versions of such a large system (*e.g.*, Linux) need to be extracted for the purpose of examining various evolution phenomena (*e.g.*, growing complexity), speed becomes a critical concern in the design of a program extractor.

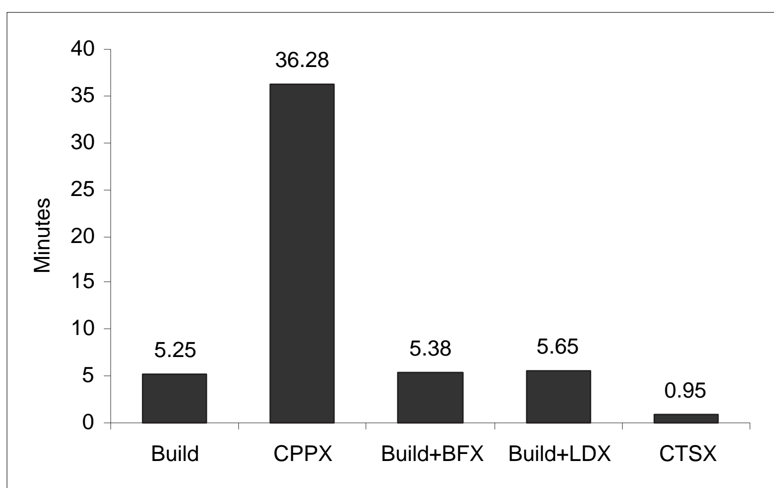


Figure 2.5: Performance comparison of the CX extractors

We applied the CX extractors on PostgreSQL 7.4 (an open source database management system) to conduct a performance comparison. Each CX extractor was used repeatedly to extract the system three times. The average extraction time was calculated. The average time required to build the system was also collected in order to provide a standard base for showing speed differences. Figure 2.5 displays the performance results obtained on a

Linux machine with an Intel Pentium IV 1.6GHz CPU and 1GB memory.

It took 5.25 minutes on average to build PostgreSQL by using the standard tool chain which includes *configure*, *make*, *gcc* and *ld*. We reused this build process to conduct program extraction by substituting CPPX for *gcc*. The total expended time was 36.28 minutes which is about 7 times large as the build time. BFX was used after build success and cost only a small fraction of the build time, roughly 2.5%. LDX spent slightly more time than BFX since it resolves cross-references between object modules. CTSX spent the least amount of time, which approximately accounts for 18% of the build time. Overall, CTSX, BFX/LDX and CPPX can have speed differences up to an order of magnitude.

A detailed examination of the four components of CTSX shows that Ctags, Ctagspost, Cscope and Cscopepost roughly account for 10%, 20%, 35% and 35% of the total extraction time respectively. Reusing Ctags and Cscope reduces the development time but also yields a negative impact on the speed of the extractor. However, building a more efficient extractor from scratch is achievable but would require a longer development time. In our work, we have focused on the rapid development of an extractor by means of adopting and enhancing existing tools.

2.4.2 Accuracy and Robustness

In CppETS (C++ Extractor Test Suite), Sim suggests that accuracy and robustness are two important dimensions for evaluating an extractor [SHE02]. Accuracy measures the correctness of facts extracted by a program extractor, and robustness measures how well the extractor deals with irregularities present in the source code.

A software system, which has been maintained over a long period of time, often contains code written in different programming languages or different dialects of the same language. Languages like SQL and Assembly may also be embedded in the code. It is not uncommon

that the system might be targeted at an outdated computing platform or built on obsolete libraries, which are not available nowadays. The missing source files (including header files) and unrelated entities with similar names make matters even worse when extraction needs to be conducted over a long history of releases. All such irregularities complicate building the program extractor. The extractor robustness and the accuracy of the extractor output must be treated properly.

The three extractors CPPX, BFX and LDX are of high accuracy and low robustness while CTSX is the opposite. A detailed explanation is given below:

- Based on the well-engineered compiler frontend of GCC, CPPX theoretically outputs data as accurate as the abstract syntax graph constructed by the compiler. However, the evaluation of CPPX against CppETS yielded a low score of accuracy due to the premature implementation of the extractor itself [SHE02].

- BFX and LDX are very close to a compiler-based extractor in terms of accuracy and robustness. Several factors contribute to this claim. First, BFX and LDX are built on mature software tools (GNU LD) and libraries (BFD) which are of product quality and have been exhaustively tested. Thus the accuracy of their output is equivalent to what a code linker or a binary utility tool (*objdump*) can see from the binary code. Second, BFX and LDX operate only after source code are compiled into object code. If the compiler failed to produce object code, it is not possible for BFX and LDX to finish the extraction. So, both extractors are only as robust as a compiler.

- Given that Ctags and Cscope are mainly based on lexical analysis and robust parsing techniques, CTSX is able to recover gracefully from unexpected syntax and continue parsing without a failure. The direct consequence is that CTSX produces more errors than CPPX, BFX and LDX. In particular, CTSX is not satisfactory in extracting

facts from large C++ systems. The use of CTSX in building an open source software evolution database without failure shows its high robustness in reality.

2.4.3 Systematic Support for C/C++ Extraction

The CX extractor suite supports a systematic extraction of C/C++ programs. By *systematic*, we mean the following:

- The CX extractors cover different steps in the build process of C/C++ a program. CTSX deals with un-preprocessed source code; CPPX works on preprocessed source code in the step of compilation; BFX is used after source compilation; and LDX is used in the step of code linking.

  In fact, the current CX suite lacks a fact extractor targeted at the preprocessing step. As a result, dependencies between macros and other source code entities cannot be extracted. A viable solution will be to instrument the GNU C preprocessor to develop a new fact extractor, thus extending the current suite to cover the entire build process. This remains our future work.

- The CX extractors can be conveniently embedded into the build process to automate program extraction without causing much interference. Especially, CPPX and LDX work as substitutes for GCC and GNU LD.

- The suite is applicable to two most important types of software artifact: source code and binary code. CPPX and CTSX extract program facts from source code but BFX and LDX extract from binary code.

2.5 Related Work

The C/C++ source code is commonly extracted using parser-based extractors, which can be handwritten from scratch or built on existing parsers or compilers (*e.g.*, GCC). Several extractors such as rigiparse [MOTU93], CPPX [CPP02], CAN [FBMG01] and TkSee/SN [TkS03] belong to this category. These extractors can produce detailed structural information. However, the full parser based approach does not solve the robustness issue (dealing with missing code or syntactical errors).

There are a number of parsing techniques based on the idea of partial extraction and regular expression matching [CC03, Moo01, MN96]. Murphy and Notkin present a lightweight lexical technique for extracting information from source code, structured data files, and documentation [MN96]. Their approach allows the user to specify language features using hierarchical patterns and regular expressions. Moonen proposes a formalism called Island Grammars for partially specifying languages that contain irregularities and generating a parser based on the specification [Moo01]. Islands are specified using production rules and regular expressions. Islands are captured by the generated parser. The rest of the source program is treated as water and ignored. An island grammar based parser generator called MANGROVE has been developed. These extraction techniques are generally more flexible, lightweight and fault tolerant. But the extractors built on them normally produce less accurate results than a full parser based extractor.

By contrast, our work is not intended to develop more complicated and advanced extraction techniques but to reuse existing tools to form a small suite of program extractors to support a wide variety of C/C++ extraction needs. We have adopted the CPPX extractor and developed instrumented versions of GNU compiler tools (*e.g.*, LD). These extractors can be conveniently embedded into the build process of a software system to extract structural information. In case that the system cannot be built successfully, CTSX can be used

to carry out the extraction. CTSX is satisfactory in terms of performance and robustness though its output is less accurate than what the other three CX extractors produce.

2.6 Conclusion

CPPX, BFX, LDX and CTSX are complementary C/C++ extractors, which cover different steps in the build process of a C/C++ software system. These extractors provide a range of tradeoffs among accuracy, robustness and efficiency. They provide systematic tool support for a wide variety of program extraction tasks. In particular, CTSX is efficient and robust for extracting a software system over hundreds of versions. The benefits of these extractors are discussed with regard to two major applications: (1) creating program comprehension pipelines and (2) building an open source software evolution database.

Chapter 3

Improving Linkage Resolution in System Model Extraction

Separate compilation and linking is critical to the development of modern C/C++ software systems up to millions of lines of code. As a consequence, it is essential to separate program extraction and program model linking in the field of reverse engineering. In other words, a fact extractor is first applied to extract program models from individual files and then a linker is used to create a software system model by combining separate program models.

This separation has diverted a significant amount of effort to developing new extraction techniques or tools but left program model linking unattended in reverse engineering. This can result in problematic extraction of software system models. Linking errors (unexpected or missing dependencies) can be introduced to the resulting system model even if individual files can be extracted correctly. Even worse, these linking errors can propagate to models at higher levels of granularity and yield an increasingly larger impact on various downstream software analysis such as reflexion modeling and software clustering.

In this chapter we describe four different methods for linking individual program models and discuss their benefits and limitations. A comparison of these methods is conducted on

an open source system, PostgreSQL. Empirical results show that inappropriate linking can lead to a relatively large number of dependency anomalies at higher levels of granularity but a small set of linking heuristics can be effective for reducing the occurrences of dependency anomaly. Independent of system configuration, these linking heuristics can be particularly useful for creating large software system models from hundreds of versions.

3.1 Introduction

Modern software systems can be extraordinarily large. For example, a large database system can have several million lines of source code. In order to analyze such large systems, it is often necessary to extract system models which have dependencies among program units at varying levels of granularity, such as the function level, the file level, and the subsystem level. Here, a dependency commonly refers to a function call, a variable access, or a macro reference between two program units such as files. This kind of system model represents a static software structure and is used in various downstream software analysis, for example, reflexion modeling [MNS95], software clustering [MMCG99, TH00a], software architecture recovery [KC98] and architecture repair [TGLH00].

A significant amount of effort has been devoted to developing novel program extraction techniques [Moo01, MN96] and tools [Aca96, CPP02, FBMG01, TkS03]. By contrast, linking separately extracted program models into a large software system model has been somewhat overlooked. There perhaps are two main reasons. First, the acquisition of system configuration information to control the linking process can be difficult for a large software project [Cal88, FSG04]. Second, people may not be aware of the extent to which linking errors can be introduced by an inappropriate model linker. As a result, the system configuration information is often ignored, and consequently model linking is simplified as

matching symbol names (strings) globally without taking into consideration the concrete system configuration.

If not controlled properly, the process of linking separate program models can produce erroneous (unexpected) or missing dependencies, which we call *linkage anomalies* in this chapter. In an extracted system model, linkage anomalies may account for a small percentage of all the resolved linkage dependencies at the lowest level of granularity. However, abstracting them into higher level dependencies can yield a much larger impact on various downstream software analysis. An illustrative example is given below.

In a case study of PostgreSQL 7.4 [Pos03], we found that linkage anomalies account for a small percentage of cross-references below the function level but they can result in relatively more higher level dependency anomalies in a progressive manner [WH04b]. The following figures are obtained used the RAW linking method, which will be further explained in Section 3.2.2. In PostgreSQL, about 1.7% of all the resolved cross-references (function calls and variable uses) below the function level are anomalies. The high-level dependency anomalies caused by these anomaly cross-references account for 3.4% of dependencies at the module level and 6.9% at the subsystem level[1]. If more types of lower level cross-references (*e.g.*, references to macros and types) are considered, about 29% of dependencies at the subsystem level are anomalies. If one performs reflexion modeling analysis on the extracted system model, (s)he will inevitably end up with many architectural divergences, which are not caused by the system implementation but by inappropriate linkage resolution.

We discuss what can cause linkage anomalies and how to reduce them in the extraction of software system models. The main idea will be centered on collecting necessary system configuration information or applying a small set of linking rules and heuristics. This work

[1]A module refers to an object module that is directly compiled from a source program file. A subsystem refers to a source code directory that directly contains at least one of the compiled modules.

complements our previous work on developing a suite of C/C++ program extractors with a focus on improving the accuracy of system models. When approaching the end of this chapter, one shall see that combining the robust CTSX extractor and a small set of linking heuristics provides an efficient means for extracting software system models of reasonable quality from hundreds of versions.

The remainder of this chapter is organized as follows. Section 3.2 explains the concept of program model linking and linkage anomaly. Section 3.3 describes a number of linking rules and heuristics which can be applied to resolve linkage anomalies. Section 3.4 describes four different methods for linking separate program models. Section 3.5 describes how to setup an experiment to compare linking methods under different conditions. Section 3.6 presents empirical results we obtained through conducting an experiment on PostgreSQL. Section 3.7 draws the conclusion.

3.2 Program Model Linking

Program model linking refers to the process of combining multiple linkable program models (LPM) into one model and resolving symbol references to appropriate definitions. One can think of a LPM as a semantic representation of an object module. A program model linker works like a code linker. However, it does not deal with addresses. Its primary functionality is to resolve cross-references from one LPM to another.

3.2.1 Sample Schema of LPM

A number of program extractors have been developed [Aca96, CPP02, FBMG01, TkS03]. Each of them outputs program data in accordance with a different schema. We adopt the sample schema shown in Figure 3.1 to facilitate LPM-related discussions. This schema

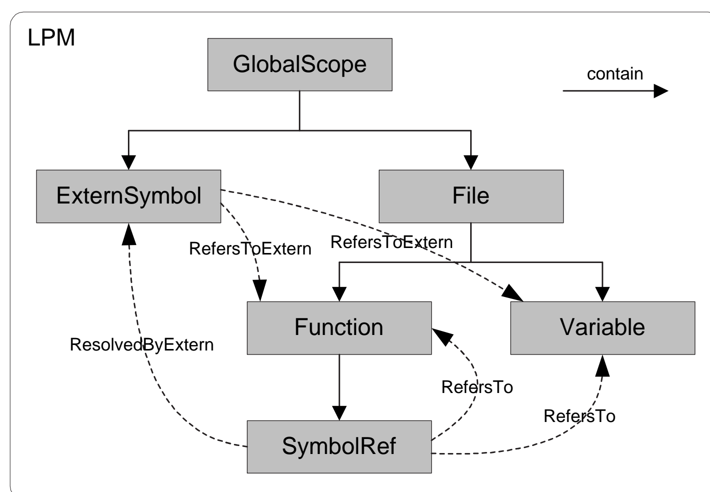


Figure 3.1: A sample schema of LPM

is a significantly reduced version of the CPPX/Datrix schema [CPP02, HHL+00]. Many abstract syntax graph (ASG) nodes such as `Type` and `Statement` have been removed for simplicity.

The `GlobalScope` represents the universal root of all LPMs. A `File` entity is a source file or an object module. Every `ExternSymbol` entity has a unique string-based identifier in the global naming space, namely `GlobalScope`. Every `SymbolRef` entity has an attribute called `kind`. In this sample schema, the `kind` attribute can take a value of `F` or `V`. The `F` value means a function call and the `V` value means a variable use.

This schema has five relations. The `contain` is a structural relation and it forms a tree for any LPM. The `ResolvedByExtern` relation represents symbol references, which need to be resolved by looking for external symbol definitions. The `RefersToExtern` means that a global definition is exported via its symbolic name. The `RefersTo` relation represents the references resolvable within a LPM. The last relation is `$INSTANCE` which stores the typing information about each entity in the LPM.

3.2.2 LPM Linking

The resolution of cross-references among a collection of LPMs can be concisely expressed as a relational composition of two binary relations: `ResolvedByExtern o RefersToExtern` where `o` stands for the operator of relational composition. This expression calculates cross-references from `SymbolRef` entities to `Function` and `Variable` entities. For example, if a `SymbolRef` entity `x` is resolved by an `ExternSymbol` `y` and `y` refers to a `Function` `z`, then `x` links to `z`. The link between `x` and `z` needs to be further checked by examining whether the `kind` attribute of `x` is `F` (function call). In addition, this link needs to be checked with regard to the configuration of the system from which the LPMs have been extracted.

Program model linking is a fundamental mechanism for deriving a model representing a software system by means of combining program models that are separately generated (or extracted) as the system is being compiled (or parsed). The obtained system model must not have any dangling symbol references among individual program models. This prohibition mandates that any symbol references resolvable within the system must be resolved. However, it is optional to resolve those symbol references pointing to the environment, in which the software system is embedded.

3.2.3 Linkage Anomalies

A linkage anomaly is an erroneous (unexpected) or a missing (unresolved) linkage dependency in a software system model. Because it is the program extractor that resolves symbol references to definitions within the same program unit such as a source file and an object module, we only discuss what can cause linkage anomalies among different program units.

Multi-Resolution

A symbol having more than one definition can cause multiple resolutions of a symbol reference to that symbol. This can occur when a software system is composed of a collection of executables and dynamic libraries. For a large system, it is highly possible that clones are used to minimize inter-dependencies among different source files. Consequently, a symbol can have more than one definition in various parts of the system. If the process of linking separately extracted program models is not properly controlled, the resolution of a symbol reference may produce multiple direct references with only one of them representing the true cross-reference dependency.

For example, PostgreSQL 7.4 has 18 executable programs and 31 dynamic libraries targeted at the Linux platform. There are 75 symbols with multiple global definitions. One of them is a function called `EncodeDateTime`, which is defined in two different source files, `datetime.c` and `dt_common.c`. The first file is used to generate the executable `postgres`, which is the backend database server. The second file is used to build the dynamic library `libecpg.so`, which provides support for embedding SQL in C. If a program model linker ignores the fact that `EncodeDateTime` has multi-definitions, every symbol reference to the name `EncodeDateTime` will be resolved to generate two direct references.

Incomplete Resolution

A possible solution to the multi-resolution problem is to leverage the software build process to constrain program model linking. The idea is to collect system configuration information to determine inter-dependencies among program files. These inter-dependencies can then be used to control a program model linker. Unfortunately, this solution does not guarantee a complete linkage resolution. A software system build often consists of dynamic libraries. When generating a dynamic library, a code linker such as the GNU linker LD leaves some

dangling symbol references which need to be resolved by the runtime linker as the dynamic library is loaded into memory.

The following describes a case from PostgreSQL. The function `tuplestore_begin_heap` defined in file `tuplestore.c` is called by function `exec_init_tuple_store` defined in file `pl_exec.c`. The file `tuplestore.c` is part of the backend server `postgres`, and `pl_exec.c` is compiled and then linked into a shared library called `libplpgsql.so`, which provides a loadable procedural language. If all the steps of building PostgreSQL are exactly followed to carry out program model linking, any symbol references to `tuplestore_begin_heap` in the file `pl_exec.c` will not be resolved. This shows that incomplete resolution can lead to linkage anomalies.

3.3 Linking Rules and Heuristics

This section describes a number of rules and heuristics useful for governing the behavior of a program model linker. They serve as a remedy means for reducing the number of linkage anomalies in an extracted software system model.

3.3.1 Linking Rules

There are two linking rules that produce no ambiguities even if only incomplete information can be obtained about the system configuration.

R1 Scope of File

Files are fundamental units for organizing modern software systems. Within the scope of a source program file (*e.g.*, `.c` and `.h` files), symbol references can be resolved accurately using current compiler techniques. For example, the CPPX source extractor based on the

GCC front end can produce abstract syntax graphs with all symbol references within the scope of a compilation unit correctly resolved. There are no linkage anomalies introduced by CPPX. For this reason, the resolution of symbol references within a file's scope is not a priority issue to be considered. One can assume that a program model extracted from a source file or a compilation unit is correct due to the proper use of parsing techniques and scoping rules.

Any symbol references that cannot be resolved within the scope of a source file (or a compilation unit) are considered to be references to external symbol definitions. Namely, their resolution will result in cross-references among source files (or compilation units). From this point forward, the discussion will only focus on the resolution of external symbol references.

R2 Unique Definition

If a symbol has a unique definition throughout the entire software system, the references to the symbol can be resolved without ambiguity. This rule is the most useful since it can resolve the majority of external symbol references.

3.3.2 Linking Heuristics

After the use of rules R1 and R2, symbol resolution can become ambiguous if no or little information is gained about the configuration of a software system. In this case, a number of linking heuristics need to be applied to improve the accuracy of a program model linker. We now describe four useful linking heuristics.

H1 File-File

If a cross-reference is present from file F1 to file F2, F2 is given higher priority when being searched for the definition of an external symbol referenced in F1. This heuristic is called the *File-File* linking heuristic. Figure 3.2 provides an illustration.

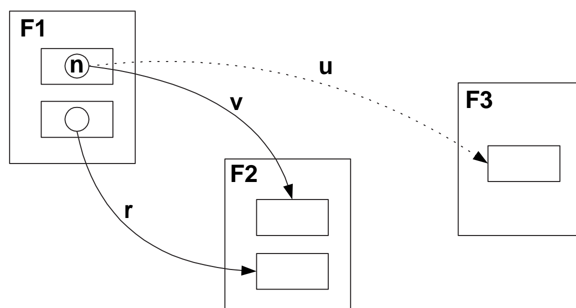


Figure 3.2: Linking heuristic: File-File

A symbol reference in file F1 to name n need to be resolved unambiguously. There are two cross-reference candidates, v and u. Since there exists a resolved cross-reference r between F1 and F2, F2 is given higher priority to be searched for the definition of symbol n. Therefore, v is determined to be a correct cross-reference in this case.

H2 File-File Closure

This heuristic can be seen as an extension of H1. The previously obtained cross-references are abstracted into file-level dependencies. The transitive closure of these dependencies is then calculated. This closure can be used to guide H1 in symbol resolution.

Figure 3.3 shows a simple case. There is no direct cross-references between F1 and F2, but an indirect dependency between these two files is formed by r1 and r2 through file Fx. Therefore, F2 is given higher priority over F3 to be searched for the definition of symbol n and u is determined to be an anomaly to eliminate.

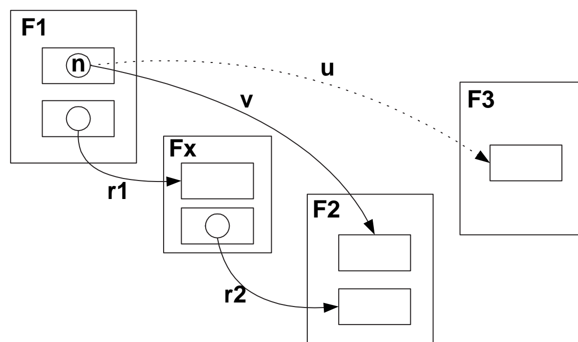


Figure 3.3: Linking heuristic: File-File Closure

H3 Same Subsystem

In the C programming language, a static definition in a compilation unit has higher priority of being linked to when a symbol reference is to be resolved in that unit, even if an external definition with the same name exists in other files. This scoping rule can be extended to be a heuristic in the resolution of symbol references within the scope of a subsystem. We refer to this heuristic as the *Same Subsystem*. Due to the nature of software organization, files from the same subsystem are generally more closely related to one another than files from different subsystems.

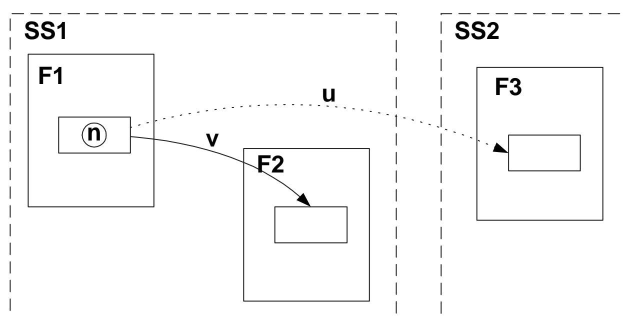


Figure 3.4: Linking heuristic: Same Subsystem

Figure 3.4 illustrates heuristic H3. The symbol `n` referenced inside `F1` is ambiguously

linked to symbol definitions in F2 and F3 respectively. Since files F1 and F2 are in subsystem SS1, it is highly possible that v is a correct cross-reference while u is an anomaly.

H4 Nearest Super Subsystem

Within a nesting subsystem hierarchy, a super (or ancestor) subsystem of a file is a subsystem which indirectly contains the file. According to H4, a nearer super subsystem of a file has higher priority to be searched than a farther super subsystem when a symbol reference in that file needs to be resolved without ambiguity.

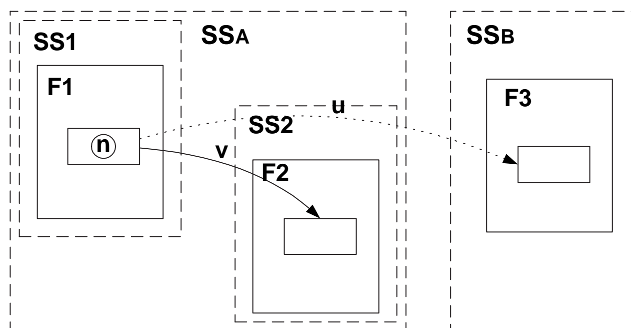


Figure 3.5: Linking heuristic: Nearest Super Subsystem

In Figure 3.5, the nearest super subsystem of F1 that contains the definition of symbol n is SSA. So, u is determined to be an anomaly.

3.4 Linking Methods

This section describes four methods for linking program models extracted separately by a program extractor. Each method uses a different strategy for applying constraints and thus produces linked system models with varying degrees of confidence. The more constraints are applied, the more accurate the resulting system model is.

3.4.1 Raw Linking

The RAW linking method is the simplest. It knows nothing about the configuration of a target software system and does not differentiate multiple external definitions of a symbol. Consequently, it does not solve the multi-resolution problem at all. This linking method calculates cross-references among a number of linkable program models using the following expressions:

```
XRef = ResolvedByExtern o RefersToExtern
XRef = Typecheck(XRef)
```

where function `Typecheck` ensures correct type match. The `XRef` is a binary relation which represents all the resolved cross-references. Besides this equation, no constraints governing the linking process will be applied. Clearly, this linking method will introduce unexpected dependencies among program units (*e.g.*, source files) if external symbols are multi-defined in the system. This method is referred to as RAW because it does nothing else but simple string match and type match.

3.4.2 Heuristics-Based Linking

The linking rules and heuristics as described in Section 3.3 can be used as a remedy means to reduce the number of linkage anomalies caused by the raw linking method. In addition, the resulting linker has no need for understanding how a software system is configured.

The following describes the implementation of a model linking algorithm based on both linking rules and heuristics. The `XRef` is initially an empty relation, and `SRef` stores the unresolved `ResolvedByExtern` symbol references. The algorithm first applies every linking rule in `Rules` to resolve symbol references and then it loops through all `Heuristics` until

no heuristic can resolve any symbol references in `SRef`. For simplicity, this implementation is referred to as HEU from now on.

```
XRef = EmptyRel;          // Cross-references
SRef = ResolvedByExtern;  // External symbol references

// Apply linking rules
for r in Rules {
    xref = r(SRef);
    XRef = XRef + xref;
    SRef = SRef - dom(xref) o SRef; // Delete resolved symbol references
}

// Apply linking heuristics
loop = TRUE;
while(loop) {
    loop = FALSE;
    for h in Heuristics {
        xref = h(SRef);
        if(#r > 0) {
            loop = TRUE;
            XRef = XRef + xref;
            SRef = SRef - dom(xref) o SRef;
        }
    }
}
```

In the above implementation, the expression `r(SRef)` represents that rule `r` is applied to `SRef` to resolute symbol references to external definitions. A similar explanation can be given to the expression `h(SRef)` where heuristic `h` instead of rule `r` is used. The expression `dom(xref)` returns the domain of relation `xref`. The `xref` stores temporarily calculated cross-references. The expression `dom(xref) o SRef` returns the symbol references in `SRef` which have been resolved. In this implementation, the executing order of linking heuristics is from H1 to H4.

When HEU is applied to link separately extracted program units of a software system, heuristics H3 and H4 require a hierarchical decomposition of that system. By default, such a decomposition can be the source folder structure in most circumstances. There are other options, such as a hierarchical clustering manually created by an expert.

3.4.3 Simulation-Based Linking

In contrast to RAW and HEU, the linking method described in this section requires some understanding of the system configuration. The more a program model linker knows about the system configuration, the fewer linkage anomalies will be introduced by the linker into the resulting system model. However, acquiring system configuration information is no less difficult than extracting facts from the source code [FSG04]. This section describes only several options for collecting configuration-related information for the purpose of improving linkage resolution.

The main idea is to tap into the build process of a software system to collect necessary information as the system is being built. The collected information is then used to control the process of program model linking. Two techniques can be used: *tool instrumentation* and *tool wrapping*. We now illustrate these techniques via some concrete examples.

Instrumenting the Code Linker

The tools for building programs such as *make*, the compiler *gcc* and the code linker *ld* can be instrumented to dump their internal data representations. We have instrumented *ld* to dump build dependencies among object modules. For example, as the `heap` subsystem of PostgreSQL 7.4, namely the `postgresql-7.4/src/backend/access/heap` source folder, is being compiled, the instrumented code linker will dump the following fact:

```
postgresql-7.4/src/backend/access/heap/SUBSYS.o:
    postgresql-7.4/src/backend/access/heap/hio.o
    postgresql-7.4/src/backend/access/heap/heapam.o
    postgresql-7.4/src/backend/access/heap/tuptoaster.o
```

This fact means that three object modules are linked into a relocatable module called `SUBSYS.o`. A program model linker can be instructed to link separately extracted program units which are related to the three object modules. If the program extractor BFX is used, a `.ta` file will be created for each `.o` file. The model linker can link `hio.o.ta`, `heapam.o.ta` and `tuptoaster.o.ta` into a partial linkable model `SUBSYS.o.ta` which corresponds to the intermediate module `SUBSYS.o`. This is just one linking step. The linking process will continue until the entire build process successfully finishes. A complete build of PostgreSQL 7.4 on a Linux platform (Linux Kernel 2.6.8 installed on a Pentium IV 1.6GHz PC) involves 99 linking steps in total. The facts dumped by the linker in all the linking steps form an acyclic dependency graph. The root nodes[2] of such a graph represent the final executables or dynamic libraries. Their associated program models need to be combined to represent the entire software build.

[2]A root of the acyclic graph is a node on which no other nodes in the graph depend.

The above example shows how separately extracted object modules can be progressively linked as a software system is being built. If separately extracted program units are source files (*e.g.*, `.c` or `.cpp`), an implicit naming convention between source files (*e.g.*, `hio.c`) and object modules *e.g.*, `hio.o` can be used to guide the program model linker. In a more complicated situation where header files (`.h`) are involved, more build dependencies among source files are needed. However, those dependencies can not be collected at the stage of code linking.

Wrapping the GCC Compiler

The compiler *gcc* has rich options for controlling the C preprocessor. One of these options is –M, which instructs the preprocessor to output the dependencies of the main source file which are readable to *make*. This feature is useful for collecting dependencies between source files and header files. A simple wrapper for *gcc* can be created to add –M related options (*e.g.*, –MD or –MMD) if option –c is found in the arguments passed to *gcc*.

For example, the –M enabled compilation of file `hio.c` from the `postgresql-7.4/src/backend/access/heap` directory dumps the following information.

```
hio.o: hio.c ../../../../src/include/postgres.h \
    ../../../../src/include/c.h ../../../../src/include/pg_config.h \
    ......
```

This fact is converted into the following form through file path canonicalization.

```
postgresql-7.4/src/backend/access/heap/hio.o:
    postgresql-7.4/src/backend/access/heap/hio.c
    postgresql-7.4/src/include/postgres.h
    postgresql-7.4/src/include/c.h
```

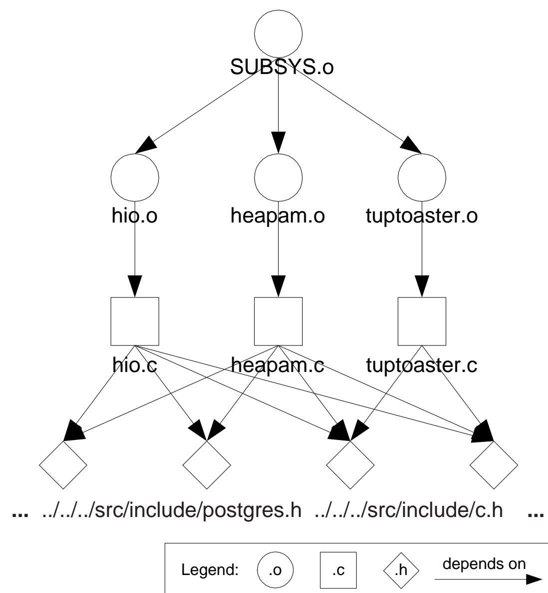


Figure 3.6: A partial build dependency graph of PostgreSQL

```
postgresql-7.4/src/include/pg_config.h
......
```

For brevity, the majority of preprocessed header files seen by `hio.c` are not listed here. This piece of information tells that file `hio.c` references symbols defined in these header files, which may include, for example, macros, data types and inline functions. A program model linker can utilize this information to resolve cross-references among the main source file and the related header files. As each individual source file is being compiled, the model linker performs a separate linking task.

Extracting the Build Dependency Graph

The collection of build dependencies among various program units such as object modules and source program files is critical to the success of a program model linker. A closer look at the dependencies collected from the build of PostgreSQL-7.4 reveals that a hierarchical dependency structure exists for C and C++ programs, in which header files (*e.g.*, `.h` and `.hpp`) are at the bottom; main source files (*e.g.*, `.c` and `.cpp`) are in the lower middle; object modules (`.o`) and archives (`.a`) are in the upper middle; and executables (`.exe`) and dynamic libraries (`.so`) are at the top. A sample dependency graph based on the facts collected for the `heap` subsystem of PostgreSQL 7.4 on a Linux PC is shown in Figure 3.6. This kind of dependency graph provides a road map for off-line linking. The model linker can perform separate linking tasks by moving progressively from bottom to top.

Summary

Through appropriate tool instrumentation and tool wrapping, the build process of software can be utilized to carry out program model linking tasks. The program model linker mimics the behavior of both the compiler and the linker throughout the entire build process. For this reason, alike methods are referred to as simulation-based linking. For brevity, they are given the name of LDM, in which "LD" stands for the linker and "M" stands for program models.

3.4.4 Simulation plus Heuristics

As discussed in Section 3.2.3, dangling symbol references may exist if dynamic (or shared) libraries are part of a system build. A LDM linker is not be able to resolve dangling symbol references by executing linking orders derived from the build dependency graph. Linking

rules and heuristics shall be used for postprocessing unresolved symbol references after the LDM linker finishes. This is a two phase linking method, *i.e.*, a sequential combination of LDM and HEU. For this reason, it is referred to as LDH.

3.4.5 Summary

Each of the four linking methods has unique characteristics as summarized below:

- **RAW** resolves symbol references without distinguishing multi-definitions of external symbols. It understands neither the configuration of a software system nor any kind of hierarchical organization of the system. Consequently, it does not solve the problem of multi-resolution. However, its benefits include simplicity and easy implementation.

- **HEU** resolves symbol references based on a set of linking rules and heuristics. This method requires a hierarchial organization of the software system, but the information related to configuration is not needed. HEU takes advantage of two important observations: (1) a software system especially a very large one commonly has a hierarchical organization based on directories, and (2) source files in the same directory are often closely related to one another. The main goal of this method is to support the mass production of software system models of reasonable quality over hundreds of versions inexpensively. By inexpensive, we mean that the system does not need to be built successfully and the configuration related data dose not need to be collected.

- **LDM** resolves symbol references by hitchhiking the build process of software through proper tool instrumentation and wrapping techniques. It mimics the behavior of the compiler and the code linker throughout the entire build process. The build relevant dependencies among program units such as object modules and source files can be dumped by various tools. These dependencies are then used to derive linking orders

to guide a linker to combine separate program models in a progressive manner. If program units can be extracted correctly, this method does not introduce any unexpected cross-references but may leave some dangling symbol references unresolved. Other drawbacks of LDM include: (1) it requires that the target system be buildable; and (2) it deals with only one configuration but not the entire system.

- **LDH** resolves symbol references by means of combining LDM and HEU. It is targeted at improving the results produced by LDM using various linking rules and heuristics. In comparison to the other three methods, LDH can produce fewer linkage anomalies but is more expensive to conduct.

An experiment has been conducted to study how different these four linking methods are from one another. The following section describes the experimental setup in detail.

3.5 Experimental Setup

The experiment is designed with two objectives in mind: (1) compare the quality of software system models derived using different linking methods, and (2) analyze the applicability of linking rules and heuristics in deriving either partial or complete software system models of reasonable quality over hundreds of versions inexpensively.

3.5.1 Experimental Conditions

The experiment is conducted on a target system under four linking conditions. Table 3.1 provides a brief summary. Each condition is described with regard to four attributes: scope, program unit, program extractor, and reference type. The scope attribute specifies whether the system model represents a particular configuration or the entire system. Two types of

Condition	Scope	Program Unit	Extractor	Reference Type
C1	configuration	`.o` file	BFX	variable use function call
C2	configuration	`.c` and `.h` file	CTSX	variable use function call
C3	configuration	`.c` and `.h` file	CTSX	variable use function call type reference macro reference
C4	entire system	`.c` and `.h` file	CTSX	variable use function call type reference macro reference

Table 3.1: Experimental conditions for linkage resolution

program unit are extracted, which are object modules (`.o`) and source files (*e.g.*, `.c` and `.h`). The used program extractors are BFX and CTSX. Four types of reference are considered, which are variable use, function all, type reference and macro reference. The type references also include references to `typedef`s.

C1 The C1 condition deals with only object modules, which can be extracted using BFX correctly. The build dependencies among object modules are collected through code linker instrumentation as described in Section 3.4. The symbol references considered are limited to function calls and variable uses.

C2 The C2 condition uses CTSX to extract source files instead of object modules. CTSX is a robust source extractor based on lexical analysis and produces facts of reasonable

quality. The source files to extract include every file needed for generating a successful build under condition C1. The build dependency graph involving object modules and source files is extracted to guide LDM and LDH. This extraction is supported by code linker instrumentation and compiler wrapping.

Besides function calls and variable uses, references to aggregate types such as `struct` and `union` are added.

C3 The C3 condition further relaxes C2 by allowing additional types of symbol reference. The references to macros and aggregate types such as `struct` and `union` are added. The extracted build dependency graph is the same as the one used under condition C2. However, condition C3 deals with part of the system which is slightly larger than the configuration seen under condition C2. This is because CTSX is used independent of the configuration of the system and it is not aware of conditional compilation.

C4 The C4 condition is the least restrictive. It changes the scope from one configuration to the entire system. Unfortunately, determining all potential software configurations is a NP-complete problem [Cal88]. It can become problematic to apply LDM or LDH on all configurations. To solve this problem, an approximate solution is approached by means of expanding the build dependency graph for one configuration in two steps. First, an automated inclusion analysis tool is applied to expand the bottom part of the graph. Dependencies among implementation files and header files are produced. Second, manual work is conducted to specify which source files are built into which modules. These two steps are relatively easy to perform because only a small portion of the entire target system need to be examined after the initial build dependency graph is extracted from one particular configuration.

As conditions change from C1 to C4, one can expect that the quality of system models produced by each linking method might decrease because the scope is expanded from one configuration to the entire system and more types of symbol reference are involved. This is indeed the case as can be seen from the experimental results described in Section 3.6.

3.5.2 Baseline for Comparison

A baseline needs to be established in order to compare system models derived using different linking methods under each condition. The system model produced by LDH appear to be the closest to a correct model with no linkage anomalies because of the following reasons:

- LDH follows stepwise linking orders derived from the build dependency graph which is constructed as the target system is being compiled;
- The linking rules and heuristics are further applied to postprocess unresolved symbol references (*e.g.*, dangling references);
- Manual work for verifying symbol references resolved by linking heuristics is expected to be in a relatively small amount.

These points help increase our confidence that a reasonable baseline model can be derived using LDH. Given a baseline, $\texttt{XRef}_{\texttt{Base}}$, the comparison is done by means of calculating the set intersection, $\texttt{XRef}_{\texttt{Base}} \cap \texttt{XRef}_{\texttt{Method}}$, and set differences, $\texttt{XRef}_{\texttt{Base}} - \texttt{XRef}_{\texttt{Method}}$ and $\texttt{XRef}_{\texttt{Method}} - \texttt{XRef}_{\texttt{Base}}$, where `Method` refers to one of the four linking methods. The former set difference denotes the cross-references which are neglected by `Method`, and the latter represents those which are produced by `Method` but are not expected.

For a linking method, the anomaly ratio (AR) with regard to a baseline model is defined as:

$$AR(\texttt{Method}) = \frac{|(\texttt{XRef}_{\texttt{Base}} - \texttt{XRef}_{\texttt{Method}}) \cup (\texttt{XRef}_{\texttt{Method}} - \texttt{XRef}_{\texttt{Base}})|}{|\texttt{XRef}_{\texttt{Method}}|}$$

The anomaly ratio can also be calculated for dependencies at higher levels of granularity such as the source file level and the subsystem level. High-level dependencies are commonly created by means of abstracting cross-references at the lowest level of granularity.

3.6 Experimental Results

This section describes the experimental results obtained from a target system, PostgreSQL. For clarity, the obtained results will be presented in two steps. The first step is solely for the purpose of demonstration and only condition C1 is involved. The second step summarizes and discusses the experimental results under all conditions.

PostgreSQL is a large open source database management system (DBMS) implemented in C. The release 7.4 has 569 `.c` files and 424 `.h` files, which approximately amount to 520 thousand of lines of code (KLOC) in total. A typical build of PostgreSQL 7.4 on the Linux platform generates 465 object modules, and each `.o` module can be mapped to a source file ending in `.c`. These object modules are used to create 18 executable programs, 31 dynamic libraries, and 50 intermediate relocatable object modules.

3.6.1 Result Analysis Under Condition C1

Table 3.2 summarizes the obtained results by using four linking methods under condition C1. The system model of PostgreSQL has 40002 symbol references to external definitions (functions and variables) in total and the same number of cross-references if all are resolved correctly. Each of the four methods produces different results. The RAW linking method

Method	Resolved	Cross-Ref.	Unresolved	Unexpected
RAW	40002	40679	0	677
HEU	39967	39967	35	2
LDM	37518	37518	2484	0
LDH	40002	40002	0	0

Table 3.2: Linkage resolution under condition C1 (PostgreSQL)

resolves all but produces 677 anomalies; HEU resolves 39967 and yields 2 anomalies, but 43 symbol references are left unresolved; LDM produces the least accurate model, in which 2484 symbol references are not resolved; and LDH produces 40002 cross-references which have been manually verified to be correct.

Table 3.3 shows the results obtained from individual steps of LDH and HEU. The first step of LDH is LDM which cannot resolve all symbol references by itself. The rule R1 and the heuristic H2 are used to achieve a complete resolution. This is caused by the creation of dynamic libraries (see Section 3.2.3). The runtime linker will resolve the 2484 dangling symbol references by means of loading the relevant dynamic libraries into memory. Using R1 and H2, these dangling references can be resolved easily. A manual examination reveals that the 6 cross-references produced by H2 are correct. Due to the nature of LDM and R1, LDH therefore correctly produces all the 40002 cross-references among 465 object modules of PostgreSQL 7.4.

Table 3.3 also shows seven individual steps in HEU. Heuristics from H1 to H4 all produce cross-references as they are applied in the first round. However, only H2 is effective in the second round. In the end, 35 symbol references are unresolved. A comparison of the cross-references produced respectively by HEU and LDH reveals that H4 introduces 2 anomaly references. This example shows that linking heuristics neither guarantee a complete linkage

LDH	Resolved	Cross-Ref.	Unresolved	Unexpected
LDM	37518	37518	2484	0
R1	2478	2478	6	0
H1	0	0	6	0
H2	6	6	0	0
HEU	Resolved	Cross-Ref.	Unresolved	Unexpected
R1	39365	39365	637	0
H1	150	150	487	0
H2	45	45	442	0
H3	301	301	141	0
H4	77	77	64	2
H1	0	0	64	0
H2	29	29	35	0

Table 3.3: Individual linking steps of LDH and HEU (PostgreSQL)

resolution nor prevent linkage anomalies.

Impact of Anomalies

We now discuss what impacts linkage anomalies have on deriving system models at higher levels of granularity. The 40002 cross-references produced by LDH are abstracted into 3772 dependencies among 465 object modules and 649 dependencies among 87 subsystems. Here, a subsystem in PostgreSQL refers to a source folder that must contain directly at least one of the 465 object modules. These high level dependencies provide a baseline for comparison because the results of LDH are correct. Table 3.4 lists the anomaly ratios obtained for each linking method at three levels of granularity.

Condition	Method	SymbolRef level AR	File level AR	Subsystem level AR
C1	RAW	1.66%	3.43%	6.89%
	HEU	0.09%	0.51%	1.88%
	LDM	6.62%	3.91%	7.81%
	LDH	0.00%	0.00%	0.00%
C2	RAW	2.13%	4.45%	6.39%
	HEU	0.28%	1.53%	2.58%
	LDM	5.62%	8.74%	18.76%
	LDH	0.00%	0.00%	0.00%
C3	RAW	21.62%	29.15%	29.03%
	HEU	1.84%	2.71%	5.09%
	LDM	8.65%	11.25%	17.65%
	LDH	0.07%	0.11%	0.15%
C4	RAW	22.42%	30.37%	28.33%
	HEU	7.22%	9.13%	15.13%
	LDM	8.35%	12.40%	25.37%
	LDH	0.27%	0.61%	1.37%

Table 3.4: Anomaly ratios at varying levels of granularity (PostgreSQL)

The 677 anomalies that are introduced by the RAW linking method result in 134 dependency anomalies at the object module level and 48 at the subsystem level. Correspondingly, the anomaly ratios are 3.43% and 6.89% respectively. In the case of LDM, the 2484 unresolved symbol references result in 142 missing dependencies at the module level and 47 at the subsystem level. These missing dependencies result in inaccuracies of 3.91% and 7.81% respectively. The 35 unresolved symbol references plus 2 erroneous references caused by

HEU lead to 18 missing and 1 erroneous dependencies at the module level and 11 missing and 1 erroneous at the subsystem level. The anomaly ratios are approximately 0.51% and 1.88% respectively.

For PostgreSQL 7.4, LDH achieves a complete and accurate linkage resolution and HEU produces a system model with much higher quality as opposed to those produced by RAW and LDM. It is interesting that LDM actually produces the models of the worst quality at all three levels of granularity. This seems rather counterintuitive at first glance. However, recalling the build of PostgreSQL has 18 executables and 31 dynamic libraries, one can see that such a highly fragmented organization of executable code is the main reason for why LDM produces more linkage anomalies than expected in PostgreSQL.

3.6.2 Result Discussions Under All Conditions

Figure 3.7 provides quantitative comparison of cross-references at three levels of granularity (the SymbolRef level, the file level and the subsystem level) in PostgreSQL under the four experimental conditions (C1 to C4). The corresponding anomaly ratios are summarized in Table 3.4. The following lists several interesting observations.

- As the most complicated linking method among the four, LDH consistently produces better results than the other three methods under all conditions. Its worst anomaly ratio is 1.37%, which is obtained at the subsystem level under condition C4.
- Due to incomplete resolution, LDM produces more dependency anomalies than RAW under conditions C1 and C2. While, RAW surpasses LDM to produce more anomalies under conditions C3 and C4. Recalling that C1 and C2 involve only function calls and variable uses but C3 and C4 handle two more types of reference (to macros and data types), we can conclude that multi-resolution outperforms incomplete resolution

Figure 3.7: Quantitative comparison of cross-references in PostgreSQL

in producing anomalies as more types of reference are considered.

- HEU is the second best among the four linking methods. A closer look at Figure 3.7 reveals that missing dependencies rather than unexpected (erroneous) dependencies account for the majority of anomalies produced by HEU. This indicates that linking heuristics described in Section 3.3 treat multi-resolution negatively. As a result, they tend to pick one or fewer references from a set of suspicious references and filter out the rest as erroneous references. Remedial heuristics need to be designed to counter this undesirable tendency. Enhancing HEU remains future work.

- In general, anomaly ratio tends to increase as the level of granularity increases. This can have an undesirable impact on various software analyses. For example, software clustering algorithms, which commonly operate on dependencies at the file level, may be sensitive to linkage anomalies and thus produce inaccurate clusterings. A user who relies on Reflexion Model to investigate structural change at the subsystem level may encounter a fairly large number of unexpected dependencies, which are not caused by the actual system implementation but by inappropriate linking.

3.7 Conclusion

Program model linking is an important step in the extraction of software system models. However, its role for improving the accuracy of a system model is somewhat neglected. As a result, a significant number of linkage anomalies (erroneous or missing dependencies) can be introduced into the resulting system model.

In this chapter, we described four methods for linking separate program models into a software system model. We compared these methods by applying them to PostgreSQL 7.4 under four conditions. Each condition has a different coverage of the system and takes into

consideration different types of reference. Empirical results have shown that heuristics can be effectively used to reduce linkage anomalies. The heuristic linking method called HEU can produce significantly better results than RAW (using neither heuristics nor system configuration) and LDM (depending only on system configuration). Even in comparison to LDH (LDM further enhanced by heuristics), HEU produces reasonably accurate models. HEU is significantly more efficient than LDH since no system configuration information is collected.

In the field of software evolution, collecting system configuration information is no less difficult than extracting program facts from the source code over hundreds of versions. In order to derive system models of reasonable quality (*i.e.*, with fewer dependency anomalies), approximation has to be used. We recommend the combination of a robust efficient source code extractor and a small set of linking heuristics as a practical means for mass production of system models over an extended period of time. In fact, we have relied on this approach to generate the system models stored in EvolDB, which we have described in Chapter 2.

Chapter 4

An Empirical Study of Punctuated Software Evolution

Theories of biological evolution provide rich sources of ideas from which software evolution research can benefit. Inspired by the theory of *Punctuated Equilibrium* [EG72] and some previous observations on discontinuous phenomena in software evolution such as ripple effect [LB85], we propose that software evolution can be viewed as punctuated equilibrium. Software systems evolve through an alternation between long periods of small changes (with little impact on the system architecture) and short periods of large avalanche changes (of architectural importance).

This chapter provides an interpretation of punctuated equilibrium with regard to software system evolution. Based on this interpretation, we conduct empirical studies on three open source systems (OpenSSH, PostgreSQL and Linux Kernel) to observe punctuated evolution from a structural perspective. The obtained results show that the three systems we examined all exhibit strong characteristics of punctuation during their evolution.

4.1 Introduction

In the field of biological evolution, the work of Darwin on the origins of species and natural selection is the most widely known [Dar59]. According to Darwin, species develop through a sequence of small variations (or mutations) and are gradually shaped by natural selection into novel species. His theory of evolution is often referred to as *gradualism*. In early 1970s, Eldredge and Gould proposed to view biological evolution as *punctuated equilibrium* [EG72]. From their point of view, species stay relatively stable over long periods of time, and sudden rapid change called punctuation causes new species to come into existence, whose fate is determined by natural selection. The controversy over punctuated equilibrium (whether it contradicts gradualism[1]?) has greatly stimulated fruitful empirical research in the field of paleontology and evolutionary biology over the past years.

Previous software evolution research has resulted in observations of discontinuous change in various forms. For example, the work of Antón and Potts on the functional paleontology of a large telephone system revealed that a burst of telephone features occurred every 8 to 12 years [AP01]. Aoyama observed significant architectural changes across generations of mobile phone systems, which he called discontinuous evolution [Aoy01]. Parnas suggested major system restructuring as one of several effective means for preventing software aging (or decay) [Par94]. Although different, these observations are relevant to the occurrences of punctuation (sudden and discontinuous change) in the evolution of software systems.

This chapter describes our effort in borrowing useful ideas and concepts from the theory of punctuated equilibrium to explain software evolution. We investigate whether software

[1]There are two common uses of gradualism: Darwinian gradualism and phyletic gradualism. Darwinian gradualism states that evolution proceeds in small variations, albeit sometimes more slowly and sometimes more rapidly. It is a mode of change and has little to do with the rate or tempo of evolution. By contrast, phyletic gradualism emphasizes that evolutionary rates are geologically slow and constant.

evolution is punctuated equilibrium by means of examining structural dependency changes over system lifetime. Our hope is that such a view and its supporting evidence can spur on further efforts (or even debates) toward developing a theoretical explanation of software evolution in a way similar to what punctuated equilibrium has done to paleontology and evolutionary biology.

The remainder of this chapter is organized as follows. Section 4.2 presents our view of software evolution modeled on the punctuated equilibrium theory. Section 4.3 provides an overview of main analysis steps we have followed to investigate software system evolution. A color coded visualization technique called Evolution Spectrograph is used in our analysis to help examine how a software system changes over time. Section 4.4 describes punctuation phenomena observed in three large open source software systems. Section 4.5 discusses the meanings of punctuated evolution and several threats to the validity of this work. Section 4.6 considers related work and Section 4.7 concludes this chapter.

4.2 Punctuated Software Evolution

According to the theory of punctuated equilibrium, systems evolve through the alternation of periods of equilibrium, in which persistent underlying structures (*i.e.*, deep structure) permit only small incremental change, and periods of punctuation, in which the underlying structures are fundamentally altered [Ger91]. This theory has three main components namely *deep structure*, *equilibrium* and *punctuation*. We now provide an interpretation of these components with regard to software evolution in the following.

4.2.1 Software Architecture

Deep structure is described by Gersick as a network of fundamental interdependent choices of the basic configuration into which a system's units are organized [Ger91]. This definition shares many similarities with that of software architecture. For example, software architecture is defined by IEEE as the fundamental organization of a software system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution [IEE00]. Garlan and Perry define that the architecture of a program comprises the components of that program, their interrelationships, and principles and guidelines governing their design and evolution over time [GP95]. Therefore, software architecture can be viewed as deep structure, which controls transitions between equilibrium and punctuation throughout the entire lifecycle of a software system.

4.2.2 Periods of Punctuation

Software architecture is the primary focus of change during periods of punctuation. Qualitative change is made to alter the architecture within a relatively compact period in order to achieve stability in the long run. Changes are often intense and occur in a short time.

For an evolving software system, its architecture must be regularly adapted to changing requirements and environment else it becomes progressively less satisfactory [LR01]. At one extreme, the architecture may be altered fundamentally. For example, an application may be transformed from a centralized architecture to a distributed architecture. At the other extreme, the architecture may be partially modified to support new uses. For example, a single platform system can be restructured to support multiple platforms by introducing a virtual operating system service layer. In each case, significant changes to the system architecture are expected to occur in a relatively short period of time.

4.2.3 Periods of Equilibrium

Within periods of equilibrium, the system architecture stays relatively stable and remains capable of accommodating forecasted changes in requirements. Changes to the system are usually small and incremental. They rarely violate the principles governing the design and evolution of the system. However, it is possible that the architecture may exhibit symptoms of gradual decay, which is caused by the accumulating effect of maintenance activities such as bug fixes and feature modifications [EGK+01].

Perry and Wolf described two architectural decay phenomena: erosion and drift [PW92]. Architectural erosion is due to violations of the architecture during incremental change. These violations tend to cause increasing brittleness of a system. By contrast, architectural drift is a slow obscure process in which the architecture floats away from its original form, thus resulting in the lack of coherence and clarity of form. The architecture needs to be evolved to counter erosion and drift. Two common strategies are used: (1) perform gradual change such as corrections and cleanups, and (2) resort to sudden massive restructuring to alter the architecture. The second strategy inevitably leads to transitions from periods of equilibrium to periods of punctuation.

4.3 Methodology

In our analysis we have conjectured that punctuated evolution can be observed by examining changes to structural dependencies at the file level and functional growth measured in number of files. There are three main reasons for having such a conjecture.

- The number of files (or modules) is one of the most widely used measures for studying software evolution [LRW+97]. New functionality is commonly indicated by the growth of the number of files and it is a major driving force for architectural change.

Working at the file level makes it relatively easy to examine the relationship between new functionality and architectural change.

- Being the highest-level abstraction of a software system, software architecture can be difficult to recover due to the lack of appropriate documentation and expertise. It is a daunting task to recover architectures manually from a large number of versions for a long lived software system. In addition, architecture recovery from scratch would be biased toward one's own understanding of what the architecture ought to be.

- Adopting the source folder structure as the subsystem hierarchy is only appropriate when a system is well organized. However, it is not unusual that many open source systems (*e.g.*, OpenSSH [Ope04a]) have a large number of files contained in only one or two directories. In such a situation, the directory structure is not suitable for creating a meaningful subsystem hierarchy.

This conjecture has a side effect. In order to determine whether an identified punctuated change is of architectural importance, we have to examine software documentation from a variety of sources including release notes, technical papers and online documentation. This may require a significant amount of manual work (mainly reading).

We now explain the main steps required for analyzing the structural evolution of software systems. In addition, a simple general-purpose technique for visualizing software evolution, which is analogous to sound spectrogram, is described in detail.

4.3.1 Analysis Overview

The analysis consists of three main steps: (1) software extraction, (2) metric collection and (3) evolution visualization.

Step 1: Software Extraction

The first step extracts program models from a historical sequence of snapshots (*e.g.*, builds or releases) of a software system. A number of source code extractors are available to use, such as CCia [Aca96], CAN [FBMG01], CPPX [CPP02] and TkSeeSN [TkS03]. Each of these extractors produces lower level references such as function calls and variable uses, which need to be abstracted into dependencies at the file level. This kind of abstraction is important for scaling up the analysis of a large system over hundreds of versions. To be simple, the file level dependency is not weighted. In other words, a dependency between files `F1` and `F2` is determined only by the existence of lower level dependencies but not by their count.

The program extractor we used is LDX, which is an instrumented version of the GNU code linker LD. LDX outputs only function calls and variable uses as the program is being compiled. LDX is reasonably robust and efficient for automating program extraction over hundreds of versions. Chapter 2 provides detailed information about LDX.

Step 2: Metric Collection

This step measures change on a per-unit basis by means of comparing program models extracted from subsequent versions. Note the data to be measured is not the extracted model but the differences between two consecutive models. Depending on the extractors used in the previous step, the extracted model can be at different levels of granularity and different measures such as Number of Functions (NOF), Cyclomatic Complexity, Fan In, and Fan Out can be computed. For example, abstract syntax graphs (ASG) produced by CPPX can be used to compute Cyclomatic Complexity on a per-function basis, and cross-references produced by LDX can be used to compute Fan In and Fan Out on a per-file basis.

We chose to measure file level dependency change using Fan In and Fan Out. For exam-

ple, if a file has an old incoming dependency removed and two new incoming dependencies added as the system evolves from release i to release $i+1$, it will be assigned a value of 3 to denote its incoming dependency change. This can be seen as Fan In measured on the delta graph obtained by contrasting two releases. Fan Out with respect to outgoing dependency change is computed in the same way. For a pair of consecutive releases, a vector of values can be obtained. A sequence of vectors can be obtained over time to characterize the entire change history of the system. These vectors are further combined into a 2D matrix to serve as the underlying data model of evolution spectrograph (see section 4.3.2).

Step 3: Evolution Visualization

This step applies appropriate visualization techniques to highlight major events and trends during the evolution of a software system. For example, one of the most important evolution analysis is to examine the system growth curve using 2D data plots [GT00, LR01]. More advanced evolution visualization techniques such as Polymetric View can also be used [LD03]. We studied the evolution history of software systems using evolution spectrograph, which is detailed in the following section.

4.3.2 Evolution Spectrograph

A spectrogram-based evolution visualization technique is used to display changes to a target software system throughout its lifetime. The technique is referred to as *Evolution Spectrograph*. Analogous to a sound spectrogram, an evolution spectrograph visually characterizes how a spectrum of software related components change over time.

Spectrograph Dimensions

A sound spectrograph (or spectrogram) provides a visual representation of the frequency content of sound and its variation in time. It is normally presented in the form of an XY graph, in which the horizontal axis X denotes the time dimension, and the vertical axis Y denotes the frequency range. The brightness of a position in the XY graph indicates the relative amplitude of the energy present for a given frequency and time. Similarly, software evolution can be characterized in terms of *time*, *spectrum*, and *measurement*, and visualized using spectrographs. Each of the three dimensions is explained below.

Time

The time dimension denotes the entire or partial lifetime of a software system. Time can be measured in two ways. First, time can be measured in units of discrete evolution events, such as software releases and source repository commits [WHH04]. Second, time can be measured in terms of fixed-length periods such as days, months and years. The unit of time needs to be chosen properly to meet the purpose of a study. For example, if one wants to analyze how the structure of a system changes over time, it is appropriate to adopt releases as time units because the system structure likely undergoes noticeable change between releases. If one is interested in studying developer activities, the unit of time based on a fixed-length period (*e.g.*, month) can be more appropriate.

Spectrum

Analogous to sound decomposition into frequency components, software system decomposition into smaller software units provides a base of measurement along the Y axis. In the spectrum of sound, frequency components are arranged into an ascending order according to their values. Similarly, software units need to be ordered based on a particular property. For example, software units (*e.g.*, files) can be ordered according to the time of creation or

modification. This ordering permits one to visualize the growth curve of a target software system, which is displayed as the upper envelope of the spectrograph (see Figure 4.2).

A target software system can be decomposed into units at varying levels of granularity, such as the subsystem level and the file level. Such a hierarchical decomposition is not the only spectrum one can use to analyze software evolution. Depending on the historical data to analyze, a spectrum can consist of components in different forms. If one is interested in analyzing developer activities in a large project, the spectrum can be an ordered arrangement of developers according to their skill levels or based on the dates they join the project. If one wants to assess the language diversity of a large application, the spectrum can be a range of implementation languages, such as Assembly, C, C++, Java, and various scripting languages, which can be ordered by their time of being added into the application. Several concrete examples of spectrum are presented in [HWH05, WHH04].

Measurement

Each component in the spectrum needs to measured over the lifetime of a software system. A variety of software metrics, such as Lines of Code (LOC), Fan In/Out of dependencies, defect density and code churn can be computed on a per-component basis. For example, the empirical studies described in Section 4.4 use Fan In and Fan Out to measure dependency change on a per-file basis.

Spectrograph Model

A spectrograph uses a matrix M as its underlying data model (see Figure 4.1). For a given spectrum, each of its components will be measured according to a particular property p. A row in the matrix stores a vector of values that characterizes the evolution history of a spectrum component. A column stores a snapshot of evolution states for all components in the spectrum at a particular time point or during a particular period. If the spectrum

contains m components (c_0, c_1, ..., c_m) and time is measured using n discrete points (t_0, t_1, ..., t_n), the matrix will have the dimension of $m \times n$. This metrics-based representation mathematically characterizes the evolution history of a software system.

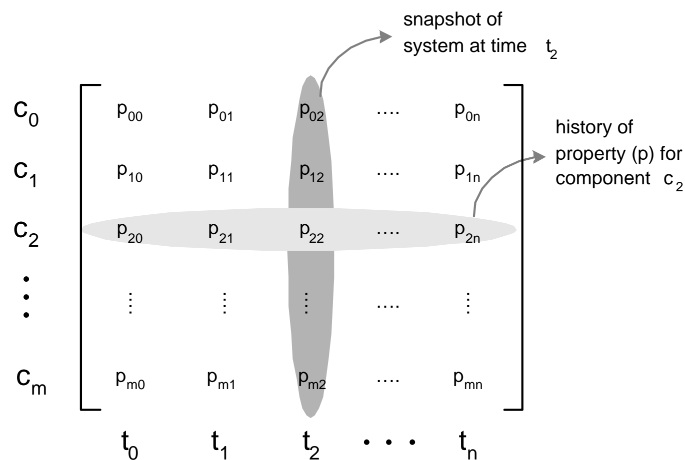


Figure 4.1: Spectrograph model based on matrix

Spectrograph Coloring

The values in matrix M are commonly coded in colors in order to produce a visual display. An appropriate coloring needs to be chosen to help the viewer easily discover evolutionary patterns embedded in the historical data characterized by M. The discovered patterns are then closely examined to gain a better understanding of the evolution of a software system. Several useful methods for coloring a spectrograph are described below.

Quartile Coloring

The quartile coloring is based on the notion of box plots. The range of values in matrix M is divided into quarters by calculating the median and the quartiles (the lower quartile is the 25th percentile and the upper quartile is the 75th percentile). Each quarter is assigned

a unique color. The quartile coloring can be extended by dividing the value range into more sub-ranges and choosing more unique colors. It has been used to study publication records from conference proceedings in the field of software engineering and change activities in large software projects [HWH05].

Gradient Coloring

A gradient coloring uses a function to map a value range to a color range. A linear gradient requires two base colors C_{min} and C_{max}, and maps the smallest value in matrix M to C_{min} and the largest value to C_{max}. Any other value in M is associated with a color determined by a linear function that maps the value range to the color range bounded by the two base colors. Other useful mapping functions such as exponential decay can also be used to color evolution spectrographs [WSHH04].

A Coloring Customized for Our Analysis

In our studies, we used a customized coloring which requires two colors C_{paint} (paint) and C_{bg} (background). A cell in the matrix is colored in C_{bg} if it does not have a value. It is colored in C_{paint} if it has a value greater than 0. Otherwise, the cell is painted in a lighter color than its previous cell in the same row. If the value at row i and column j in matrix M is denoted as $M[i][j]$ and its associated cell color denoted as $C[i][j]$. The spectrograph is rendered according to the following equation.

$$C[i][j] = \begin{cases} C_{bg} & \text{if } M[i][j] \text{ is not a value} \\ C_{paint} & \text{else if } M[i][j] > 0 \\ C_{paint} & \text{else if } j = 0 \\ lighter(C[i][j\text{-}1]) & \text{else} \end{cases}$$

Where function $lighter(c)$ converts color c into a lighter color of the same hue by decreasing saturation and increasing brightness. We chose the color dark-green for C_{paint} in the study. The generated spectrographs will be shown in Section 4.4.

<table>
<tr><td>Target System</td><td>
<table>
<tr><th></th><th>Domain</th><th>Period of Dev.</th><th>Size(KLOC)</th><th>#Releases</th></tr>
<tr><td>OpenSSH</td><td>Protocol</td><td>10/1999–03/2004</td><td>22–70</td><td>60</td></tr>
<tr><td>PostgreSQL</td><td>DBMS</td><td>01/1997–12/2003</td><td>185–525</td><td>85</td></tr>
<tr><td>Linux Kernel</td><td>OS</td><td>06/1996–07/2003</td><td>674–5141</td><td>76</td></tr>
</table>
</td></tr>
<tr><td>System Model</td><td>Scope
The system model represents only part of the system, which consists of files needed for a successful build on the Linux platform.
Dependency graph
File-level dependencies are created by means of abstracting lower level cross-references extracted by LDX.</td></tr>
<tr><td>Spectrograph</td><td>Time
Time is measured using historical releases or monthly builds.
Spectrum
Spectrum consists of object files (<code>.o</code>) ordered by the creation time of corresponding source files (<code>.c</code>).
Measurement
Fan In and Fan Out on a per-file basis. Both measures are applied to delta graphs between paired consecutive versions.</td></tr>
</table>

Table 4.1: Setting for studying punctuated software evolution

4.4 Case Studies

We have examined three open source software systems (OpenSSH, PostgreSQL and Linux Kernel) in an attempt to search for empirical evidence of punctuated evolution. Table 4.1 summarizes the empirical study setting. All three target systems are open source with each of them from a different application domain. OpenSSH is the smallest system, which has grown from 22 KLOC to 70 KLOC. Linux is the largest system, which has more than five

million lines of code up to July 2003. PostgreSQL is approximately an order of magnitude larger than OpenSSH and smaller than Linux respectively.

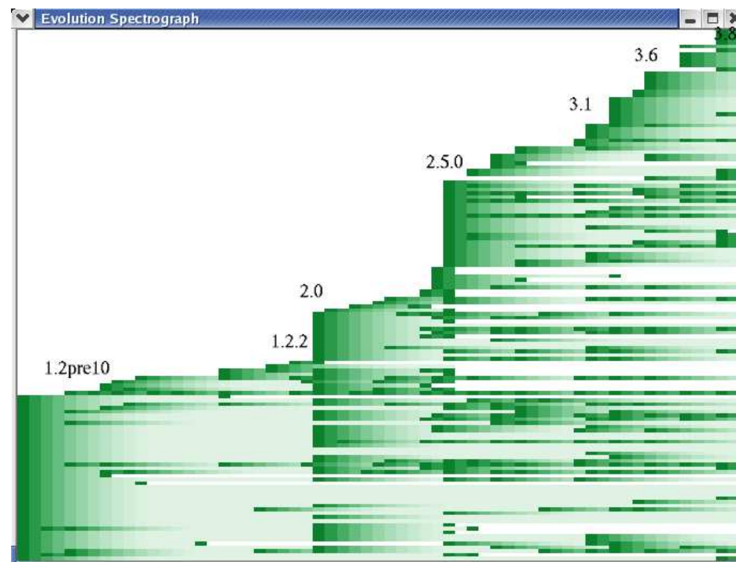
Changes to incoming and outgoing dependencies are measured for each object file over time. Two evolution matrices based on Fan In and Fan Out are created respectively for each target system. The matrices are converted into spectrographs to highlight conspicuous system-wide change events throughout the lifetime of each target system.

4.4.1 OpenSSH

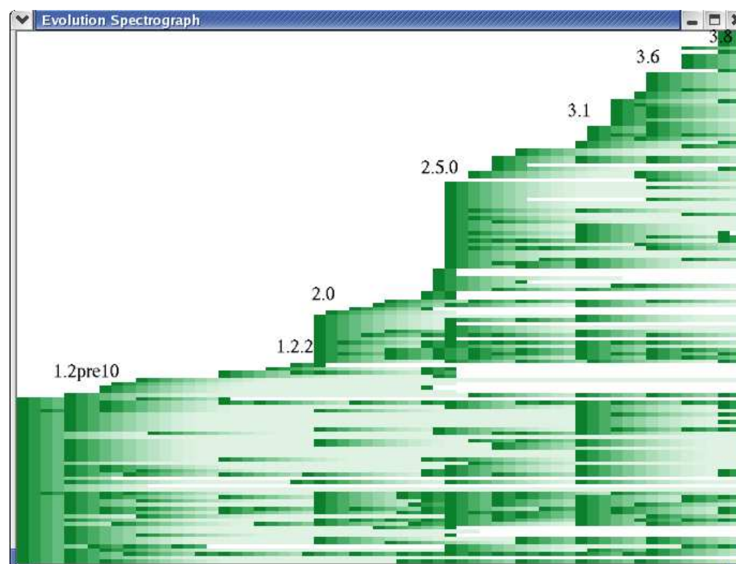
OpenSSH is a well-known open source implementation of the Secure Shell (SSH) protocol suite of network connectivity tools [Ope04a]. OpenSSH encrypts communication traffic in order to effectively eliminate eavesdropping, connection hijacking, and other malicious network attacks.

Figure 4.2 presents two spectrographs of OpenSSH, in which changes to dependencies at the object file level are plotted against release numbers. The major release numbers are displayed so that the reader can correlate them with conspicuous change events in the form of vertical bands (dark colored). Changes to OpenSSH were not evenly distributed during its lifetime. For example, release 2.0 and release 2.5.0 involved system-wide changes while the ten releases between them were relatively stable. This indicates that changes made to OpenSSH were not incremental. They occurred in system-wide bursts instead.

The figure suggests there are three periods of punctuation in the evolution of OpenSSH, which are highlighted as dark vertical bands close to releases 2.0, 2.5.0 and 3.1 respectively. The online documentation relevant to OpenSSH was examined to verify this observation. In May 2000, the developers of OpenSSH implemented support for the SSH2 protocol and release 2.0 was delivered. This major change maps to the first period of punctuation. Given that OpenSSH is a protocol-centered application, it is not surprising that this update (SSH1



(a) Incoming dependency change



(b) Outgoing dependency change

Figure 4.2: Evolution spectrographs of OpenSSH

to SSH2) resulted in system-wide changes. In November 2000, the developers implemented the Secure File Transfer Protocol (SFTP) client and server support which was shipped with release 2.5.0 in February 2001. This functionality enhancement can be associated with the second period of punctuation. The third period of punctuation near release 3.1 seems not related to functional improvement for the number of files does not show noticeable growth.

The requirements for new functionality serve as the main driving force of change during the first two periods of punctuation near releases 2.0 and 2.5.0 respectively. During each of these two periods, the size of OpenSSH, which is measured using the number of files, shows substantial growth. By contrast, the system does not grow significantly during the third period of punctuation, but its internal system structure undergoes substantial change. As the milestone transition occurs from release 2 to release 3, the architecture of OpenSSH is changed to support its future evolution.

One can gain insight into the causes of conspicuous change events by means of contrasting the spectrographs for both incoming and outgoing dependency changes. For example, if one only looks at Figure 4.2(b), it seems that release 1.2pre10 is aggressively restructured. However, Figure 4.2(a) shows this restructuring is actually caused by adding or eliminating dependencies to seven files, which are indicated by seven thin horizontal lines in a relatively dark color. A further examination of the source code reveals that two logging utility files `log-client.c` and `log-server.c` are merged into a single file called `log.c`. This merge results in widespread changes in the extracted dependency graph but not in the real source code. This phenomenon can be explained as change to one "aspect" of the system, namely, logging.

4.4.2 PostgreSQL

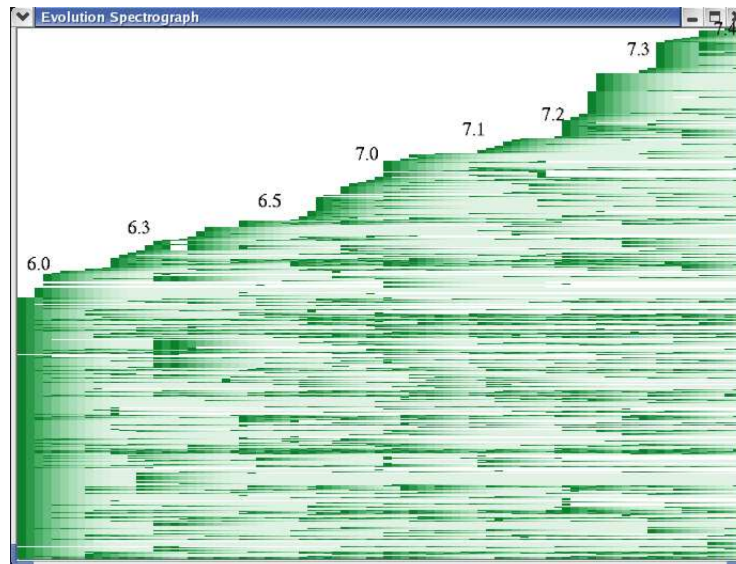
PostgreSQL is a large Object-Relational Database Management System (DBMS) [Pos03]. The development of PostgreSQL first started at the University of California at Berkeley in 1996. It soon became an open source project with a globally distributed development team. The 85 releases studied are monthly builds checked out from the CVS repository but not publicly announced official releases.

Figure 4.3 shows two spectrographs of PostgreSQL. The growth of PostgreSQL is approximately linear in terms of the number of files. One can notice that conspicuous vertical bands appear in Figure 4.3(b) but not in Figure 4.3(a). In addition, all vertical bands are close to major releases, such as 6.0, 6.3, 6.5, 7.0, 7.2 and 7.3. This suggests that each system-wide perturbation, highlighted by a vertical band in Figure 4.3(b), was actually related to a small number of files that had their incoming dependencies substantially changed. This is an interesting phenomenon that needs further investigation.

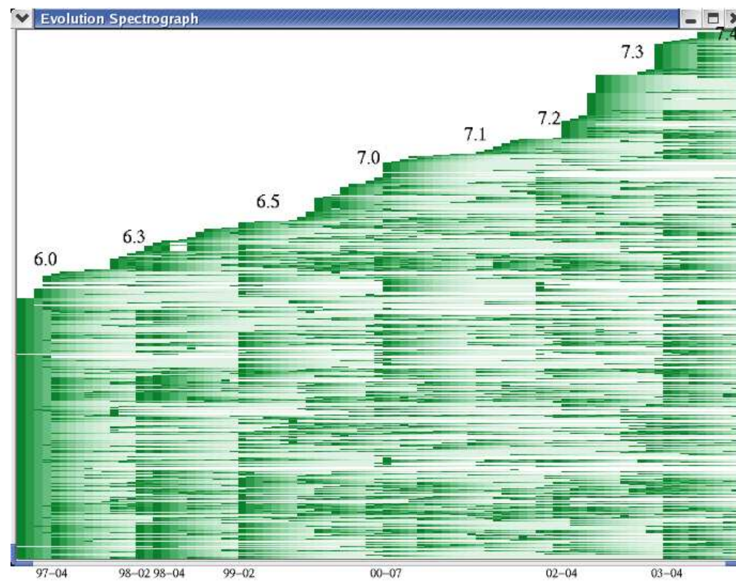
Month	#Files	Case A	Case B
Apr 1997	300	50.6%	13.0%
Feb 1998	311	52.1%	11.3%
Apr 1998	322	59.9%	15.5%
Feb 1999	336	61.6%	16.4%
Jul 2000	376	50.5%	32.7%
Apr 2002	384	39.8%	29.8%
Apr 2003	451	45.9%	5.3%

Table 4.2: Ratio of changed files in PostgreSQL

The ratio of changed files for each of the seven months that correspond to vertical bands in Figure 4.3(b) is calculated. Two kinds of change are considered: (A) files are considered



(a) Incoming dependency change



(b) Outgoing dependency change

Figure 4.3: Evolution spectrographs of PostgreSQL

to be changed only if their outgoing dependencies are changed; (B) files are considered to be changed if their dependencies to non-`utils`[2] subsystems are changed. Table 4.2 provides a summary of the results obtained. Both July 2000 and April 2002 have very high change ratios in both cases. By contrast, the other five months have much lower change ratios in case B as opposed to case A. A further examination shows that changes of dependencies to several services provided by the `utils` subsystem accounted for the majority of dependency changes. These services include error reporting and logging (`error`), memory management (`mmgr`), and cache utility (`cache`). This indicates that the developers of PostgreSQL have continually devoted efforts to several quality attributes such as reliability and performance. These quality improvements are critical to the success of a database application in a highly competitive market.

PostgreSQL's evolution exhibits characteristics of punctuation. In addition, seven periods of punctuation are approximately evenly distributed throughout the system's lifetime. Recalling that the system has grown approximately at a linear rate, one can conclude that PostgreSQL's evolution is well controlled and coordinated. Nakakoji et al. have described PostgreSQL as service-oriented software with six Core Members forming a council-like development control team [NYN+02]. Any new features or improvements first exist only as patches for a relatively long period of time, and are incorporated into the core version only after they are approved by these core members.

[2] We refer to the `src/backend/utils` directory as the `utils` subsystem, which provides basic services such as data types, error handling, memory management, data caching, and etc. Any directories that are contained by the `utils` directly or indirectly belong to the `utils` subsystem. All other directories belong to the non-`utils` subsystems.

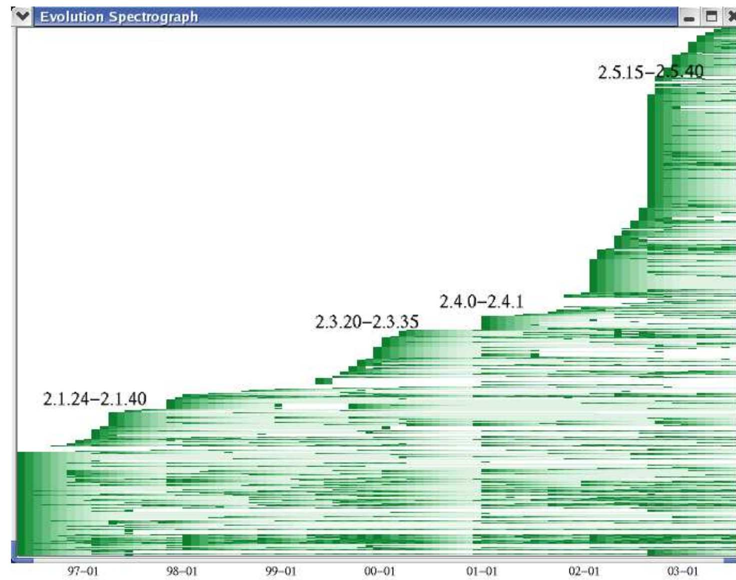
4.4.3 Linux Kernel

Linux is a clone of the UNIX operating system, originally written from scratch by Linus Torvalds and subsequently worked on by hundreds of other developers [Lin04]. The first official release, version 1.0, came out in March 1994. Linux has been evolving along two parallel paths: the stable kernel for production use and the development kernel for experimentation. By convention, the middle number in a kernel version indicates to which path it belongs: an even number indicates a stable version (*e.g.*, 2.0.7), and an odd number indicates a development version (*e.g.*, 2.1.15).

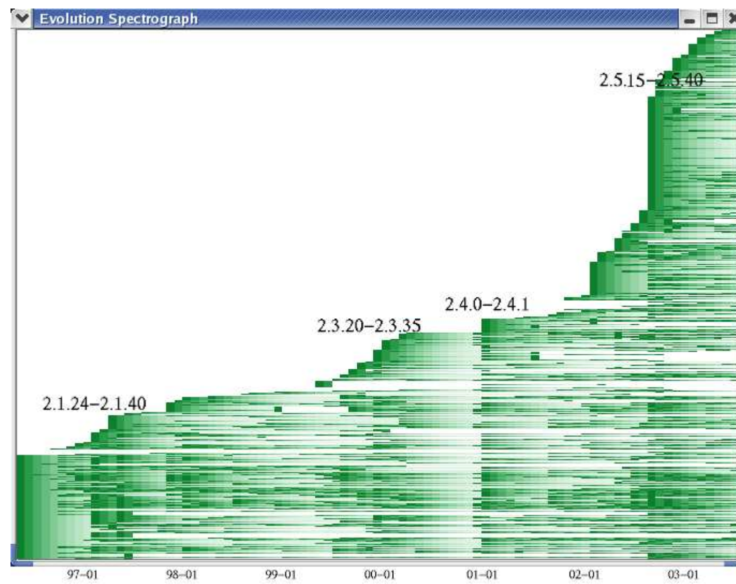
This case study of Linux involves 76 releases, from 2.0 to 2.5.75, which cover 7 years of development in total. These releases are distributed approximately on a monthly basis and ordered according to the release dates. Two spectrographs are created to characterize how the kernel evolved. The dark-colored vertical bands shown in Figure 4.4 suggest that the evolution of Linux shows characteristics of punctuation.

The vertical band associated with releases 2.4.0 and 2.4.1 indicates a punctuation caused by substantial change to the internal structure. As there are no official kernel releases from June 2000 to December 2000, this punctuation is apparently caused by the accumulating effects of six months of development. In addition, there is no considerable system growth in release 2.4.0. This is not normal given that Linux has been growing exponentially [GT00]. Linux experienced critical structural change as the milestone release 2.4.0 was delivered.

The other vertical bands are related to new functionality added to the kernel, indicating three periods of functional punctuation respectively. The first functional punctuation spans about three months, covering 17 development versions, 2.1.24 – 2.1.40. These versions were enhanced for multi-processor support and tuned for faster speed [Goo00]. The second functional punctuation involves development versions 2.3.20 – 2.3.35. Three major subsystems `fs`, `net` and `kernel` experienced substantial change. For example, several



(a) Incoming dependency change



(b) Outgoing dependency change

Figure 4.4: Evolution spectrographs of Linux

Journaling File Systems such as XFS from SGI and JFS from IBM were introduced; the network subsystem (`net`) was split into two main layers: network address translation and packet filtering; the use of the global Big Kernel Lock (BKL) was minimized by replacing it with local subsystem spin locks; and Symmetric Multiprocessor Processing (SMP) was enhanced to support up to 32 CPUs. The third functional punctuation occurs with versions 2.5.15 – 2.5.40. It was aimed at supporting high end enterprise servers and embedded devices [Lin04, San03]. For example, the threading model was improved; a new scheduler algorithm was introduced to support high loads and a large number of processors; and a unified device model was created to manage various device drivers. All these changes are critical to the Linux kernel.

4.5 Discussions

The investigation of three open source software systems shows empirical evidence for punctuated software evolution. In other words, software systems evolve through an alternation between long periods of small incremental change having little impact on the architecture and short periods of sudden discontinuous change of architectural importance.

As observed in the evolution of three target systems, periods of punctuation are mainly associated with notable system growth (new functionality) and massive system restructuring accompanying major or milestone releases. This suggests that new functional requirements and internal system structure decay are two main driving forces behind punctuation. This observation strongly confirms Lehman's first law of software evolution termed as *continuing change* [Leh97]. This law states that software systems in real worlds must be continually adapted to meet the changing environment and requirements else they become progressively less satisfactory. However, our view of software evolution puts more emphasis

on system wide restructuring as sudden intermittent occurrences with long periods of small change to the system structure in between.

4.5.1 Threats to Validity

There are several threats to the validity of this work which are described below.

- Punctuation is perceived by means of observing dependency changes among (object) files. No higher-level software organization units (*e.g.*, subsystems and components) are involved. This may over-emphasize the architectural importance of dependency changes to files within the same subsystem or component. As a result, some observed periods of punctuation may not be of great importance in the sense that the system architecture is not significantly altered at higher levels of granularity, *i.e.*, levels above object files.

- The dependency graphs extracted from each target software system have only object files, which were generated by the default build procedure comprising build tools such as *configure*, *make* and *gcc* on the Linux platform. Therefore only part of the system is examined. The ratio of `.o` files to `.c` files is approximately 97% for OpenSSH, 80% for PostgreSQL and 20–50% for Linux. There may exist some periods of punctuation which are not observed during the lifetime of these systems (particularly Linux).

- The dependencies among object files are created by means of abstracting lower-level function calls and variable uses. The resulting graph may possibly leave out some important aspects of C programs since cross-references to other important programming constructs of C such as data types and macros are simply neglected. However, using other types of cross-reference may potentially lead to more or even fewer observations of punctuation.

4.6 Related Work

Our work is mainly related to two kinds of research in the field of software evolution: (1) understanding software evolution dynamics and (2) designing novel visualization techniques to assist researchers in evolution exploration.

4.6.1 Evolution Understanding

Lehman's seminal work on OS/360 revealed that the growing entropy of local changes was counteracted by periodic global effort [LB85]. The cost and unreliability of software changes in OS/360 rose sharply during a series of minor releases and then re-stabilized through the adaptation of the architecture every few major releases. His studies were mainly based on observing system growth. By contrast, we examined punctuated evolution in open source software by tracing and visualizing structural change. All the three systems we examined exhibit characteristics of periodic global structural adaptation.

Antón and Potts studied the evolution of telephone systems [AP01]. Their study was based on longitudinal analysis of feature growth. They observed that a burst of telephone features occur every 8 to 12 years. By contrast, our work relies on evolution spectrograph to locate punctuated structural change rather than feature growth. We have observed more frequent occurrences of punctuated change, roughly every 1 or 2 years. This is perhaps because our analysis is at a lower level of abstraction as opposed to their feature analysis. Furthermore, the three open source systems in our analysis are much younger than the telephone system they investigated.

Aoyama have observed that discontinuous evolution occur in mobile phone software systems across multiple product lines [Aoy01]. During the first three generations, the architecture of mobile phone systems changed from a closed and vertically integrated

communication-centric style to an open and horizontally integrated computation-centric style. The author studied system growth and system level architectural style change. The architecture of a commercial system is commonly documented in corporate environments though it may not be up to date. Thus, documented architectures can be contrasted to identify major change in the evolution of the system. However, such architectural documentation is generally not available for open source systems. Our work thus examined structural change recovered from the system implementation. Another notable difference is that the three systems we studied are all open source.

Barry *et al.* have shown that different software systems may exhibit a similar lifetime evolution pattern which is determined using an ordinal measurement of software volatility with three dimensions: amplitude, periodicity and dispersion [BKS03]. Their approach computes an ordinal value for every snapshot of a software system during its entire lifetime, while our approach computes an evolution matrix to characterize how a spectrum of entities change over time. Their study were performed on program modification records recovered from source control repositories. We have focused on structural information extracted from a historical sequence of versions or snapshots.

4.6.2 Evolution Visualization

The evolution spectrograph is a technique modeled on sound spectrogram, and it is designed to visualize a sheer volume of evolutionary data. There is a considerable amount of related work in the field of software evolution visualization.

Gall and Jazayeri have used *percentage bars* to show three kinds of evolutionary entities (structure, attribute and time) simultaneously in one view [GJR99]. Their objective is to find conspicuous change events from the software release history to assist in future restructuring. The evolution spectrograph combines system growth, system change and

release history simultaneously. Our intention is to gain a better understanding of the paradigm of software evolution by looking for evidence of punctuated change.

Lanza proposed *Evolution Matrix* as a means to recover the evolution of object oriented software systems [Lan01]. In an evolution matrix view, a class is represented using a box with the width determined by the number of instance variables (NIV) and the height determined by the number of methods (NOM). The layout and shape are used to highlight interesting change patterns over time. The evolution spectrograph can be seen as a simplified version of his approach. The visual elements considered include only color but not shape in order to handle the sheer volume of data. For thousands of files which change over hundreds of versions, shape is only of limited use. In addition, files in the spectrum are sorted according to their creation time so that the upper envelope of the spectrograph can reflect the system growth curve. The viewer can thus correlate functional growth with punctuated change in relative ease.

Eick *et al.* have developed a set of tools for visualizing several data classes, such as code version history and release differences [BE96]. In their approach, lines of code are displayed as lines of color-coded pixels on screen in order to achieve a higher information density. The time dimension (*e.g.*, the history of releases) is also encoded into color pixels. This kind of view helps the viewer examine the recent change history of an individual file (or a limited number of files). The distortion of the time dimension into color pixels makes the view not appropriate for studying punctuated change over a large number of files.

Taylor and Munro proposed to use *revision towers* to visualize the change history stored in source control repository [TM02]. A tower-like view is created for an individual source file to show all the revisions of the file and the relationships between revisions and source releases. All towers are then displayed in a grid formation to fill the available display area, ordered according to the time of file creation. A revision tower provides very detailed

information about changes to a single file. However, such detail about individuals makes it difficult to correlate changes among different files far away from one another. For example, a temporal change spread into a large number of files cannot be easily recognized using revision towers. On the contrary, an evolution spectrograph sacrifices local individual detail for easy global perception.

Collberg *et al.* have developed a graph-based system for visualizing software evolution, called GEVOL [CKN+03]. The evolution history of a software system is characterized as a sequence of graphs and each graph reflects the system state at a given point in time. These graphs are overlayed one above another and animated over time. The differences between graphs are coded in colors to indicate how the system changes over time. GEVOL uses advanced layout algorithms in order to preserve the viewer's mental map. GEVOL is useful for visualizing how software structures evolve, such as call graphs, control graphs, and inheritance graphs. The evolution spectrograph does not deal with software structures directly but uses metrics instead. It can be customized to show how measured properties change over time.

4.7 Conclusion

This chapter described an empirical study of software evolution from a perspective based on punctuated equilibrium. Sudden and discontinuous structural changes are observed during the evolution of three open source systems. A simple scalable visualization technique called Evolution Spectrograph is used to aid in exploring punctuated changes throughout system lifetime.

The following chapter continues on studying software system evolution from a structural perspective. We will resort to automated software clustering techniques to help recover

changes to the system design so that we can perform a large scale automated analysis of punctuated evolution at the system design level.

Chapter 5

Clustering Comparison in the Context of Software Evolution

To aid software understanding and maintenance tasks, a number of software clustering algorithms have been proposed to automatically partition a software system into meaningful subsystems or clusters. However, it is unknown whether these algorithms produce similar meaningful clusterings for similar versions of a real-life system under continual change and growth. An automated algorithm of such capability can be used to support the analysis of the evolution of system design. Comparing clustering algorithms in the context of software evolution is thus needed in order to choose an appropriate algorithm.

This chapter describes a comparative study of six software clustering algorithms. We applied each of the algorithms to subsequent versions from five large open source systems. We conducted comparisons based on three criteria respectively: stability (Does the clustering change only modestly as the system undergoes modest updating?), authoritativeness (Does the clustering reasonably approximate the structure an authority provides?), and extremity of cluster distribution (Does the clustering avoid huge clusters and many very

small clusters?).

The six studied algorithms exhibit distinct characteristics. For example, the clustering from the most stable algorithm bears little similarity to the as-implemented structure of the system; and the clustering from the least stable algorithm has the best cluster distribution. The obtained results indicate that current automated clustering algorithms need significant improvement to provide continual support for maintaining large software projects and that they are not suitable for empirical software evolution analysis.

5.1 Introduction

A well documented architecture can improve the quality and maintainability of a software system. However, many existing systems often do not have their architecture documented. Moreover, the documented architecture becomes outdated and the system structure decays as rapid changes are made to the system to meet market pressure [EGK+01, Par94]. A high rate of turnover among developers can make the situation even worse. The maintenance of architectural documentation is one of many problems that confront today's large software projects. Software clustering holds out a promise of helping in this task [MMCG99, TH00a].

Software clustering refers to the decomposition of a software system into meaningful subsystems or clusters. It plays an important role in understanding legacy software systems [MOTU93], assisting in their architectural documentation [BHB99, MMCG99], and supporting their re-modularization [Sch91, Wig97]. For example, a misplaced procedure or file can be automatically identified by an intelligent clustering tool and relocated to an appropriate subsystem to reduce unexpected dependencies and to prevent the decay of the system architecture [LS91].

Ideally a software clustering algorithm should be automated to provide continual sup-

port throughout the lifetime of a large software system. In such a context, the algorithm must produce meaningful clusterings in a stable manner. To be meaningful, the algorithm should produce clusterings that can help developers understand the system. An algorithm which places all source files from a system into two large clusters will not be helpful to developers analyzing the code. To be stable, the algorithm must produce clusterings which do not change abruptly from one version to the next. An algorithm which produces clusterings of significant difference even though no major code restructuring occurred between versions is likely not to be used by developers.

This chapter describes a comparative analysis of six automated clustering algorithms, which are used to cluster subsequent versions of five large open source systems. The main goal is to clarify to what extent software clustering algorithms can be used to support remodularization and architectural documentation during the life cycle of a software system. Three criteria are used to evaluate the usefulness of these algorithms: *C1*. When a system changes modestly, the clustering produced by an algorithm should also change modestly; *C2*. An automatically produced clustering should approximate the clustering produced by an authority such as an architect; and *C3*. Automatically produced clusters should generally not be either huge (*i.e.*, containing hundreds of source files) or tiny (*i.e.*, containing very few source files). An automated clustering algorithm meeting the three criteria can be used to recover design changes from the lifetime history of a system. The recovered changes can be useful for analyzing the design evolution of the system.

The remainder of this chapter is organized as follows. Section 5.2 provides an overview of five systems which are chosen for the experimentations. Section 5.3 explains the experimental setup in the context of software evolution. Section 5.4 describes a simple ordinal measure for comparing a collection of data series. Section 5.5 describes the experimental results obtained using ordinal evaluation techniques. Section 5.6 discusses several interesting

System	Language	#Versions	#Source Files	Size (KLOC)	Graph Data
Ruby	C	73	90 – 261	74 – 187	6.5MB
KSDK	C/C++	70	21 – 1156	3 – 263	82.5MB
OpenSSL	C	73	593 – 845	164 – 278	74.4MB
PostgreSQL	C	73	771 – 947	182 – 519	99.1MB
KOffice	C/C++	70	1358 – 3266	272 – 962	1235.5MB

Table 5.1: Software systems chosen for the clustering experimentation

observations. Section 5.7 considers related work and Section 5.8 concludes this chapter.

5.2 Target Systems

In the evaluation of clustering algorithms, we choose to apply them to a number of real-life systems which are all open source software and hence available for study. Table 5.1 provides a summary of key properties of five software systems. They represent distinct application domains and have gone through a number of years of development. The experimentation covers a five year period (1999 to 2004) and versions are collected for each target system on a monthly basis. Due to a CVS update problem the last three monthly versions of KSDK and KOffice are not available at the time of data collection. Each target system is briefly described below.

1. **KSDK** is a software development kit for the K Desktop Environment (KDE) [KDE04]. It offers powerful framework support for various kinds of KDE applications.

2. **KOffice** is an integrated office suite for KDE. This suite consists of 12 major applications which are KWord, KChart, KSpread, KPresenter, Kivio, Karbon14, Krita, Kugar, KPlato, Kexi, KFormular and Filters [KOf04].

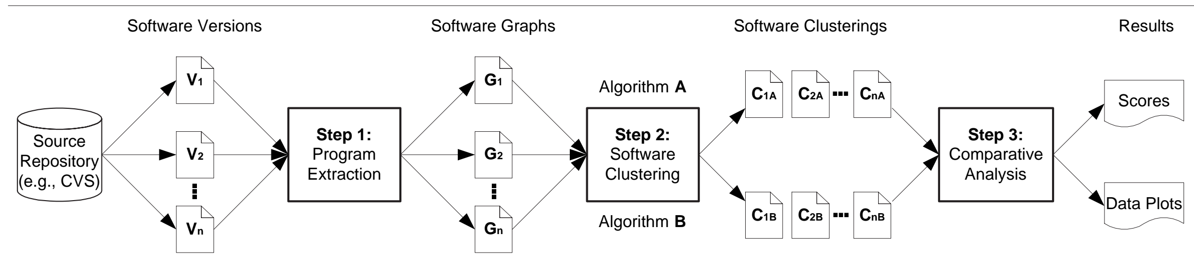


Figure 5.1: Comparison of clustering algorithms in the context of software evolution

3. **OpenSSL** is a cryptography toolkit implementing Secure Socket Layer (SSL) and Transport Layer Security (TLS) network protocols as well as several related cryptography standards [Ope04b].

4. **PostgreSQL** is a large SQL-compliant object Relational Database Management System (DBMS) which originated at the University of California at Berkeley in 1996 [Pos03].

5. **Ruby** is an interpreted scripting language designed for quick and easy object oriented programming [Rub04]. It has many convenient features for text file processing and system management.

5.3 Experimental Design

Figure 5.1 illustrates the experimental design. The source control repository (such as CVS) of a target system (such as PostgreSQL) is the starting point. Monthly versions called V_1, V_2 and so on are retrieved from the repository. In step 1 (Program Extraction), we extract a directed graph G_i from each version V_i. In the graph each node represents a source file in the target system. Each edge represents a static dependency (such as a reference to a

variable or a data type, a call to a function, or a call to a macro) from one file to another. In step 2 (Software Clustering), we run a number of clustering algorithms called A, B, and etc. on each graph G_i, producing clusterings C_{iA}, C_{iB}, etc. Finally, in step 3 (Comparative Analysis), we rely on a set of criteria to evaluate the algorithms. The three steps are now detailed below.

Program Extraction

In the first step cross-reference graphs are extracted from 70 to 73 monthly versions of each target system. The source extractor we chose is CTSX which is built on Ctags [Cta04] and Cscope [Csc04]. CTSX uses Ctags to extract program entities (*e.g.*, functions, variables, macros and data types) and Cscope to retrieve references (*e.g.*, function calls) to entities found by Ctags. These static cross-references are then lifted to the level of files to produce directed edges between nodes (files) in the extracted graph. This lifting to the level of files greatly decreases the size of the graph. It is an important consideration when dealing with many versions of a large software system.

Software Clustering

In the second step a collection of clustering algorithms is run over graphs extracted from each monthly version of each target system. A sequence of clusterings is produced for each target system by each algorithm.

We chose a set of six clustering algorithms based on their availability and their discussion in the literature. We limited our choice to available implementations that run in batch mode since we needed to run each of these algorithms many times as they were applied to different target systems over a large number of versions. We chose software clustering tools created by researchers Anquetil, Mancoridis and Tzerpos. Anquetil designed a hierarchical

clustering algorithm suite which offers a selection of association and distance coefficients as well as update rules [AL99]. Four algorithms from this suite were chosen and given names of the form of CL** and SL** where ** encodes the parametrization, as described below. Mancoridis and Mitchell provided us with their Bunch suite [MMCG99]. Tzerpos provided us with his algorithm for comprehension driven clustering (ACDC) [TH00a]. We now give a brief description of these clustering algorithms.

1. **CL75** is an agglomerative clustering algorithm based on the Jaccard coefficient and the complete linkage update rule [AL99]. The cut-point height for the dendrogram is set to 0.75. A higher cut-point will result in a smaller number of clusters in the resulting dendrogram.

2. **CL90** has the same configuration as CL75 except that its cut-point height is set to 0.90 [AL99].

3. **SL75** is an agglomerative clustering algorithm based on the Jaccard coefficient and the single linkage update rule [AL99]. The cut-point height is set to 0.75.

4. **SL90** has the same configuration as SL75 except that its cut-point height is set to 0.90 [AL99].

5. **ACDC** is an algorithm based on program comprehension patterns and it attempts to recover subsystems that are commonly found in manually-created decompositions of large software systems [TH00a]. The patterns used in ACDC include source file, directory structure, body-header, leaf collection, support library, central dispatcher and subgraph dominator.

6. **Bunch** provides a suite of algorithms that include Hill Climbing, Exhaustive, and Genetic Algorithms [DMM98, MMCG99]. We tried three different hill climbing con-

figurations: NAHC (nearest ascend hill climbing), SAHC (shortest ascend hill climbing), and a customized configuration with the minimum search space greater than 55% and the randomized proportion of the search space equal to 20%. We only discuss the third configuration because it produces similar results as NAHC and SAHC. For brevity, the third configuration is referred to as Bunch from this point forward.

Comparative Analysis

In the third step the chosen clustering algorithms are compared based on three criteria, *C1*, *C2* and *C3*, which are briefly mentioned in Section 5.1. We now detail the three criteria:

C1 Stability

Similar clusterings should be produced for similar versions of a software system. This criterion emphasizes the persistence of the clustering structure of successive versions of an evolving software system. Under conditions of small and incremental change between consecutive versions, an algorithm should be stable, *i.e.*, it should produce similar clusterings for successive months.

C2 Authoritativeness

Clusterings produced by an algorithm should resemble clustering from some authority. An authoritative clustering may be manually created by a human architect. It may also be automatically derived from the directory structure of the target system. In the latter case, authoritativeness is tantamount to adherence to the source folder structure.

C3 Extremity of Cluster Distribution

The size distribution of clusters within a clustering should not exhibit extremity. In particular, a clustering algorithm must avoid two extreme situations: (1) the majority

of files are grouped into one or few huge clusters (sometimes called *black holes*); and (2) the majority of clusters are singletons or tiny clusters of 2 or 3 files (forming what are sometimes called *dust clouds*). In either situation the clustering algorithm cannot be useful in practice.

The detailed information about comparisons based on each criterion will be presented in Section 5.5.

5.4 A Simple Ordinal Measure

To support the comparative analysis of clustering algorithms over consecutive versions of a target system, a simple ordinal measure for ranking a number of data series needs to be defined. A data series DS is a sequence of quantitative values such as <1.2, 2.32, 1.78>. For series DS_i and DS_j, functions $Above(DS_i, DS_j)$ and $Below(DS_i, DS_j)$ are defined as:

$$Above(DS_i, DS_j) = \frac{|\{n \mid DS_i[n] > DS_j[n], 1 \leqslant n \leqslant |DS_i|\}|}{|DS_i|}$$

$$Below(DS_i, DS_j) = \frac{|\{n \mid DS_i[n] < DS_j[n], 1 \leqslant n \leqslant |DS_i|\}|}{|DS_i|}$$

Series DS_i and DS_j can be seen as lines of points. Function $Above(DS_i, DS_j)$ denotes the proportion of the line formed using DS_i above the other line formed using DS_j. A similar explanation can be given to *Below*. We say that DS_i is above DS_j if $Above(DS_i, DS_j) > Above(DS_j, DS_i)$.

Given K data series, DS_1, DS_2, ..., DS_K, we have the following equations for measuring

the relative position of DS_i with regard to all the K data series.

$$Above(DS_i) = \sum_{j=1}^{K} Above(DS_i, DS_j) \tag{5.1}$$

$$Below(DS_i) = \sum_{j=1}^{K} Below(DS_i, DS_j) \tag{5.2}$$

In the next section these two equations will be adapted accordingly to obtain an ordering of clustering algorithms with regard to the three criteria (stability, authoritativeness and extremity) respectively.

5.5 Empirical Results and Interpretation

5.5.1 Stability Comparison

Tzerpos defined a stability measure based on the ratio of the number of "good" clusterings to the total number of clusterings produced by a clustering algorithm for a system [TH00b]. A clustering obtained from a perturbed version of the system is considered to be good if the MoJo dissimilarity between that clustering and the one obtained from the original version of the system is not greater than 1% of the total number of resources (source files). It can be difficult to determine a proper threshold in the context of real-life software evolution and 1% seems too optimistic in reality.

In this comparison we instead derived a relative ordering of different algorithms. Also, we evaluated an algorithms' stability using three ordinal values: High, Medium and Low. The latter method is referred to as the *HML ordinal evaluation*.

Figure 5.2 explains how to create a similarity sequence through intra-sequence comparisons. Given a sequence of n clusterings, a sequence of $n-1$ similarity values denoted as

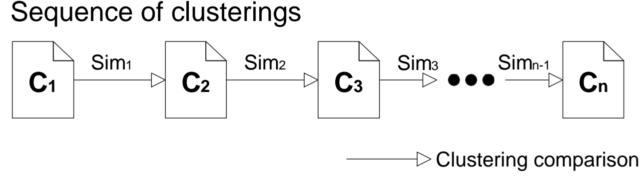


Figure 5.2: Intra-sequence clustering comparison

$<Sim_1, Sim_2, ..., Sim_{n-1}>$ can be obtained. A similarity or dissimilarity value is calculated by comparing two consecutive clusterings using a similarity measure. For example, MoJo produces dissimilarity values [TH99] while EdgeSim produces similarity values [MM01]. For non-reflexive similarity measures such as MoJo, comparisons needs to be performed in the direction a target system evolves. MoJo is chosen for this comparison for its simplicity. Correspondingly, Sim_i is defined to be $MoJo(C_i, C_{i+1})$ where $1 \leqslant i \leqslant (n-1)$.

A measure for relative stability

Given a target system S and K clustering algorithms, A_1, A_2, ..., and A_K, $MJS(A_i, S)$ is used to denote the sequence of MoJo values calculated based on intra-sequence comparison over the sequence of clusterings obtained by A_i from S. A relative score of A_i over A_j is defined as:

$$Score_{MJ}(A_i, A_j, S) = Below(MJS(A_i, S), MJS(A_j, S))$$

The relative score of A_i over all the K clustering algorithms is define as:

$$Score_{MJ}(A_i, S) = Below(MJS(A_i, S)) \tag{5.3}$$

where $Below$ is defined as Equation 5.2 and MJ stands for MoJo. Algorithm A_i is more stable than A_j with regard to system S if $Score_{MJ}(A_i, A_j, S) > Score_{MJ}(A_j, A_i, S)$. When comparing K algorithms, Equation 5.3 is used to calculate a stability score for each algo-

rithm. The K obtained scores are then sorted to determine a relative stability ordering of K algorithms. The greater the score, the more relatively stable an algorithm.

A HML-based ordinal measure

The system growth curve is denoted as SG. Two separate data series $SG10$ and $SG30$ can be obtained from SG. $SG10$ is proportional to SG and it accounts for 10% of the number of source files for each data point in SG, and correspondingly $SG30$ accounts for 30%. The series $SG10$, $SG30$, and SG divide the area below SG into three smaller regions. If the MoJo series $MJS(A_i, S)$ has at least 80% of its data points below $SG10$, the stability of A_i is H (High). If $MJS(A_i, S)$ has at least 80% below $SG30$, A_i has a score of M (Medium). Otherwise, A_i has a score of L (Low).

$$HML_{MJ}(A_i, S) = \begin{cases} H & \text{if } Score(MJS(A_i, S), SG10) \geqslant 0.8 \\ M & \text{elsif } Score(MJS(A_i, S), SG30) \geqslant 0.8 \\ L & \text{otherwise} \end{cases} \tag{5.4}$$

This equation measures how stable an algorithm is with regard to a target system rather than simply states which one is more stable than another. If a clustering algorithm has a score of H, it means that the number of MoJo operations (*i.e.*, moves and joins) is less than 10% of the total number of source files in the target system for at least 80 out of 100 transformations of consecutive clusterings.

Plots of MoJo series

Figure 5.3 shows a plot of MoJo series which were obtained from PostgreSQL for each of the six chosen clustering algorithms. In the figure, the MoJo values associated with Bunch are greater than the corresponding MoJo values associated with other algorithms. Because MoJo is a dissimilarity measure, Bunch is less stable than the other five algorithms. The

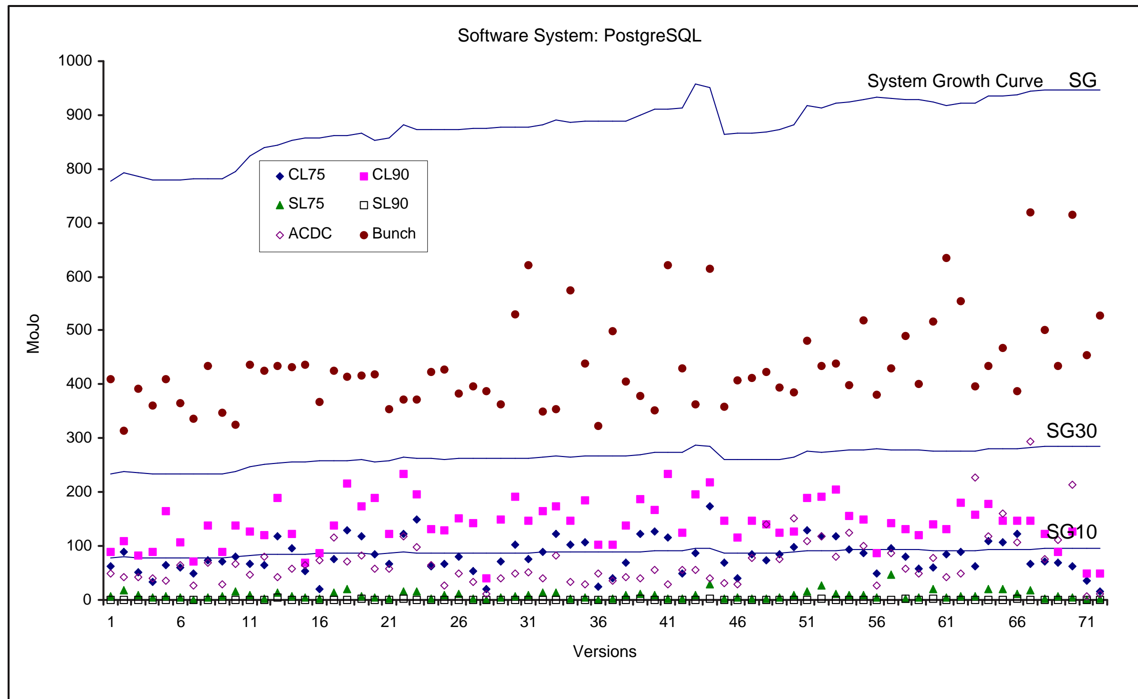


Figure 5.3: Stability comparison wrt PostgreSQL

MoJo series for NAHC and SAHC were also examined. They exhibit a similar behavior as the series for Bunch. They are omitted from Figure 5.3 for brevity.

The MoJo series associated with CL90, CL75 and ACDC reside in the middle of Figure 5.3. They appear intertwined at some locations. It can be roughly seen that ACDC appears more stable than CL75 and CL90. SL90 and SL75 are at the bottom of the figure. They are the most stable algorithms.

Ordinal evaluation

When data series are highly intertwined it can be difficult to obtain an ordering of several clustering algorithms based on visual inspection. A quantitative method is needed. Using Equation 5.3, relative stability scores can be calculated for each studied algorithm with

	KSDK	KOffice	OpenSSL	PostgreSQL	Ruby	ALL
Bunch	0.07	0.00	0.00	0.00	0.01	0.02
CL90	1.00	1.06	1.40	1.08	1.35	1.18
CL75	2.01	1.97	2.46	2.29	1.93	2.14
ACDC	2.49	3.35	2.08	2.63	2.56	2.62
SL75	3.30	3.62	3.82	4.00	3.35	3.62
SL90	4.26	5.00	4.51	5.00	3.96	4.55

Table 5.2: Relative stability scores obtained using MoJo

	KSDK	KOffice	OpenSSL	PostgreSQL	Ruby	ALL
Bunch	L	L	L	L	L	L
CL90	M	M	M	M	M	M
CL75	M	M	M	M	M	M
ACDC	M	H	M	M	H	H
SL75	H	H	H	H	H	H
SL90	H	H	H	H	H	H

Table 5.3: HML-based stability scores obtained using MoJo

regard to each target system as well as the concatenation of all five target systems. When comparing algorithms with regard to a concatenation of different systems, the similarity series from all those systems are concatenated for each clustering algorithm and then Equation 5.3 is applied. The obtained scores are given in Table 5.2. Those obtained using the concatenation are listed in the `ALL` column which shows the overall stability ordering.

All the scores in Table 5.2 tell the same story. The chosen algorithms are ordered in the direction of increasing stability as follows: Bunch, CL90, CL75, ACDC, SL75 and SL90. There is an exception with regard to OpenSSL where CL75 appears to be more stable than

ACDC since the former scored 2.46 but the latter scored only 2.08 (see Table 5.2). The cells shaded in light gray indicates this disagreement.

We also conducted the HML ordinal evaluation. The obtained HML scores are given in Table 5.3. These scores indicate that SL90, SL75 and ACDC are highly stable algorithms but Bunch is not.

To rule out the possibility that the obtained stability ordering may be biased due to the use of MoJo, we measured the ordering of these algorithms using other similarity measures, which include EdgeMoJo [WT04], EdgeSim and MeCl [MM01]. We found that EdgeSim strongly agrees with MoJo but the other two measures slightly disagree with MoJo on the ordering of CL75 and ACDC and they rank CL75 as relatively more stable than ACDC. Given that these results were obtained using evolution data from several hundred versions of real-life systems and further verified by multiple similarity measures, the following stability ordering reflects the reality in general.

Low	Medium	High
Bunch	CL90 CL75	ACDC SL75 SL90

5.5.2 Authoritativeness Comparison

Unfortunately, a stable algorithm may not produce meaningful clusterings at all. At one extreme, an algorithm producing only singleton clusters is stable over time but not useful. At the other extreme, an algorithm grouping all the files into one cluster is obviously stable but not meaningful at all. A clustering algorithm of practical use should produce clusterings similar to authoritative decompositions of a software system by experienced software engineers [BHB99]. However, an agreed upon architecture or authoritative decomposition often does not exist. This makes it difficult to do authoritativeness comparison.

We used a simple technique to create authoritative clusterings. Our technique comprises four steps: (1) create the subsystem hierarchy based on the directory structure; (2) relocate every header file (`.h`) to the subsystem that directly contains the related implementation file (`.c`); (3) merge a subsystem with its parent if it contains less than five files; (4) create a flat clustering with each subsystem in the remaining hierarchy as a cluster. For well structured software systems like KOffice and PostgreSQL, this technique is likely to produce a clustering which conforms to the mental model of the developers of the system. It can be easily automated to cluster a large number of versions.

We performed inter-sequence comparisons of clusterings from two parallel sequences. One sequence comprises clusterings produced by the algorithm in analysis and the other contains authoritative clusterings obtained using the technique described above. Figure 5.4 shows how inter-sequence comparisons are done over time. The similarity Sim_i is calculated as $MoJo(C_i, C_{iA})$. Based on the obtained similarity series $<Sim_1, Sim_2, ..., Sim_n>$, we carried out authoritativeness comparison using the scoring methods for stability comparison (see Equations 5.3 and 5.4).

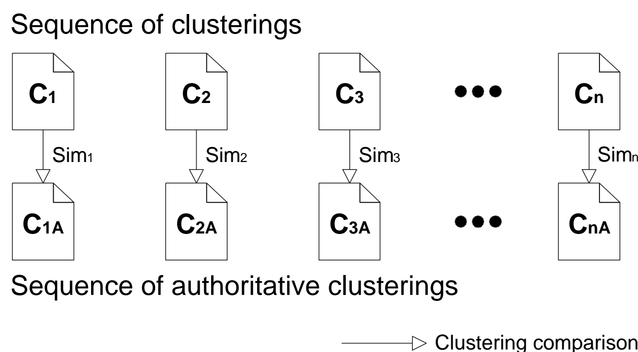


Figure 5.4: Inter-sequence clustering comparison

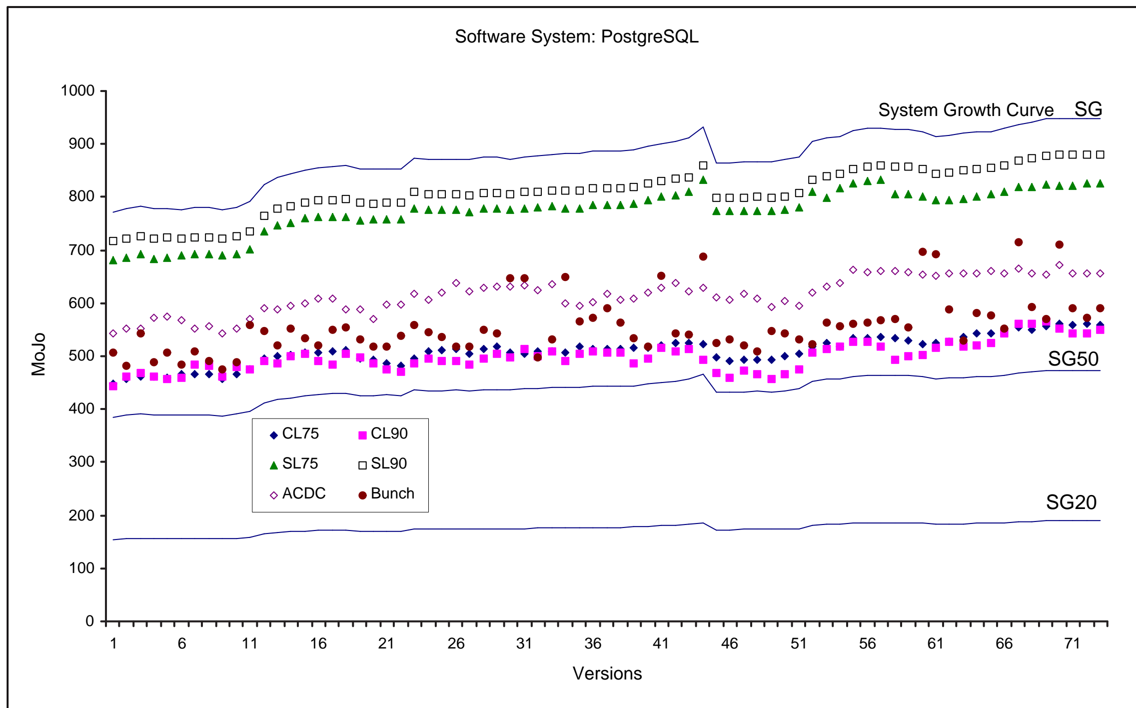


Figure 5.5: Authoritativeness comparison wrt PostgreSQL

Plots of MoJo series

Figure 5.5 shows the MoJo series obtained from PostgreSQL. At the top of the figure are two MoJo series associated with SL75 and SL90. Their values are approximately equal to the number of source files. This indicates that nearly every source file in PostgreSQL needs to be moved or joined in order to transform the clusterings produced by SL75 and SL90 to the corresponding authoritative clustering. SL75 and SL90 produce clusterings bearing little similarity to the implemented structure of PostgreSQL.

The MoJo series for ACDC and Bunch reside in the middle of Figure 5.5, with the former staying mostly above the latter. It indicates that ACDC is slightly less authoritative than Bunch. However, none of them produces clusterings resembling the implemented structure

	KSDK	KOffice	OpenSSL	PostgreSQL	Ruby	ALL
SL90	0.07	0.00	0.00	0.00	1.85	0.39
SL75	1.06	1.00	1.00	1.00	1.29	1.07
ACDC	2.79	2.06	2.11	2.12	1.55	2.12
Bunch	3.86	4.03	3.66	2.92	3.26	3.54
CL75	2.61	4.03	4.62	4.16	2.19	3.56
CL90	4.43	3.84	3.56	4.77	4.27	4.18

Table 5.4: Relative authoritativeness scores obtained using MoJo

of PostgreSQL. Roughly speaking, 60-70% of all the source files in PostgreSQL need to be moved or joined in order to transform the clusterings produced by Bunch or ACDC to their authoritative counterparts. This may be due to the fact that clusterings produced by Bunch or ACDC are too coarse (containing few clusters) than their corresponding authoritative clusterings. The most authoritative algorithms are CL75 and CL90 located at the bottom of Figure 5.5, which are slightly better than Bunch and ACDC. We manually examined the plots obtained from other target systems and confirmed the above observations.

Ordinal evaluation

The obtained relative authoritativeness scores are provided in Table 5.4. The greater the score, the more the clusterings produced by an algorithm resemble the implemented system structure, and consequently the more authoritative the algorithm turns out to be. The scores from the `ALL` column determine the overall ordering. The cells colored in light gray indicate the disagreement between the overall ordering and the one obtained from a target system. In the HML evaluation, $SG20$ and $SG50$ instead of $SG10$ and $SG30$ are chosen in order to relax the requirements on how closer a clustering should resemble the implemented

structure. All the six algorithms are ranked Low. For this reason the obtained HML scores are omitted. The final ordering is shown below. None of these algorithms is satisfactory in producing clusterings that approximate the implemented structure of a target system. This ordering is further verified using similarity measures EdgeMoJo, EdgeSim and MeCl. These measures only disagree with MoJo on the ordering of CL75 and CL90.

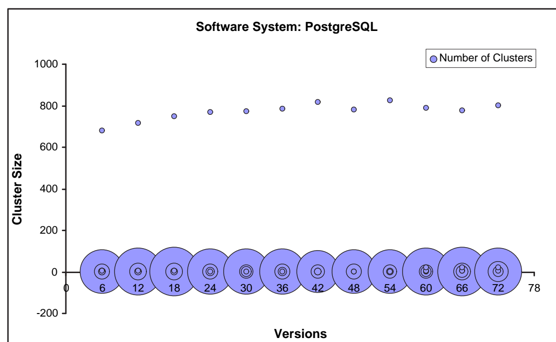
Low					
SL90	SL75	ACDC	Bunch	CL75	CL90

5.5.3 Extremity Comparison

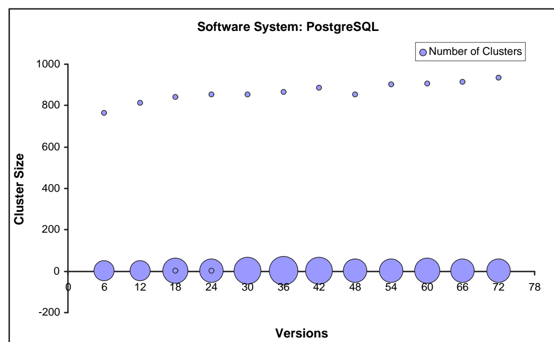
We studied the size distribution of clusters over time in an attempt to examine whether a particular clustering algorithm avoids generating huge clusters (black holes) or many very small clusters (dust clouds).

Figure 5.6 displays six bubble charts. In each chart the X axis represents versions, the Y axis represents the size of the cluster, and the size of the bubble denotes the number of clusters of the same size. For example, the clustering obtained using SL90 from version 72 of PostgreSQL has 14 clusters, in which 13 clusters are singletons and one huge cluster comprises 934 source files. This clustering is represented using two bubbles in Fig. 5.6(b), located at the coordinates (72, 1) and (72, 934) respectively. The larger bubble is 13 times big as the smaller one though the latter is a black hole representing a cluster of 934 files. Although the axes X and Y may have different scales, all the bubbles share the same scale of measurement.

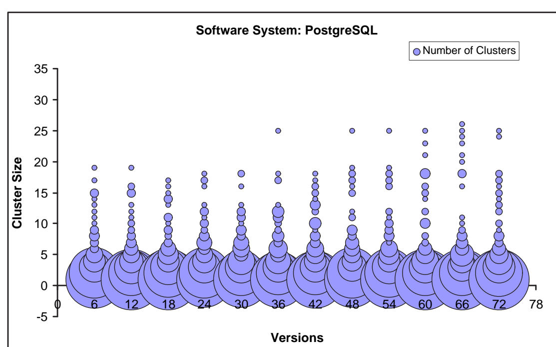
Figure 5.6 shows distributions for every sixth version in order to avoid the overlapping of bubbles in the X direction. Each algorithm exhibits a distinct cluster distribution pattern. Figures 5.6(a), 5.6(b) and 5.6(f) show that SL90, SL75 and ACDC tend to produce extreme



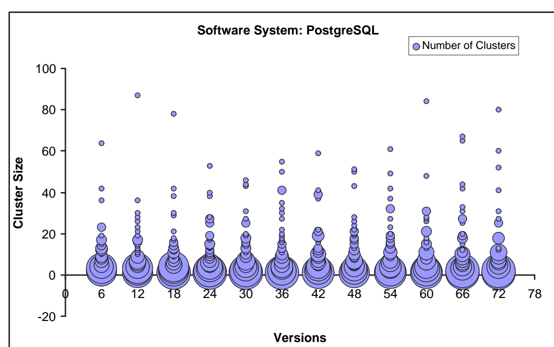
(a) Distributions obtained using SL75



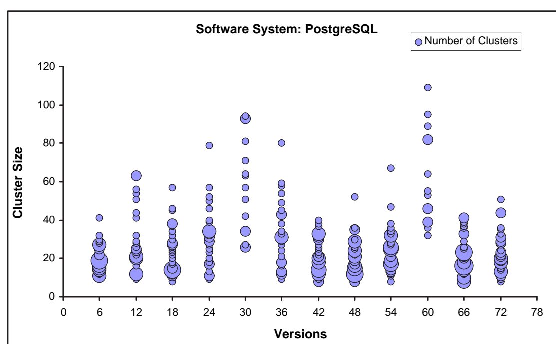
(b) Distributions obtained using SL90



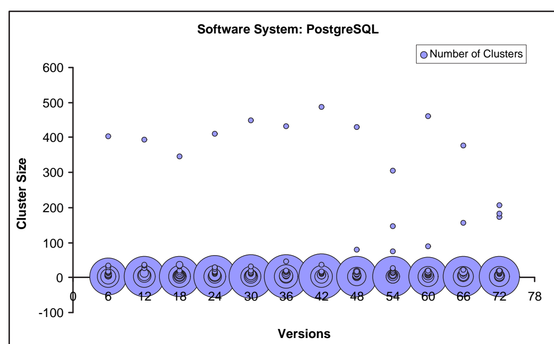
(c) Distributions obtained using CL75



(d) Distributions obtained using CL90



(e) Distributions obtained using Bunch



(f) Distributions obtained using ACDC

Figure 5.6: Distribution comparison of clustering algorithms wrt PostgreSQL

clusters which are either huge or very small. By contrast, Bunch, CL75 and CL90 produce more evenly distributed clusters. However, CL75 has a tendency of generating a relatively large number of singleton clusters as shown in Fig. 5.6(c).

A measure for non-extreme cluster distribution

In order to quantitatively evaluate the extremity of cluster distribution, we define a simple measure called NED (non-extreme distribution). Any clusters of size less than 5 or greater than 100 are considered to be extreme clusters. This is a reasonable assumption since very few clusters in the obtained authoritative clusterings contain less than five or more than a hundred source files.

NED is defined as the ratio of the number of files in non-extreme clusters to the total number of files in the target system. A larger NED value indicates a better distribution. Figure 5.7 shows several NED series obtained from PostgreSQL. The NED series obtained from KSDK, KOffice, OpenSSL and Ruby have similar patterns. So they are omitted for brevity. Bunch has the best distribution of clusters. SL75 and SL90 always group the majority of source files into extreme clusters. The NED series obtained for the other three algorithms remain in the middle of the figure.

To perform NED-based ordinal evaluation, two scoring methods are defined as:

$$Score_{NED}(A_i, S) = Above(NS(A_i, S)) \tag{5.5}$$

$$HML_{NED}(A_i, S) = \begin{cases} H & \text{if} \quad Score_{NED}(NS(A_i, S), P75, S) \geqslant 0.8 \\ M & \text{elsif } Score_{NED}(NS(A_i, S), P50, S) \geqslant 0.8 \\ L & \text{otherwise} \end{cases} \tag{5.6}$$

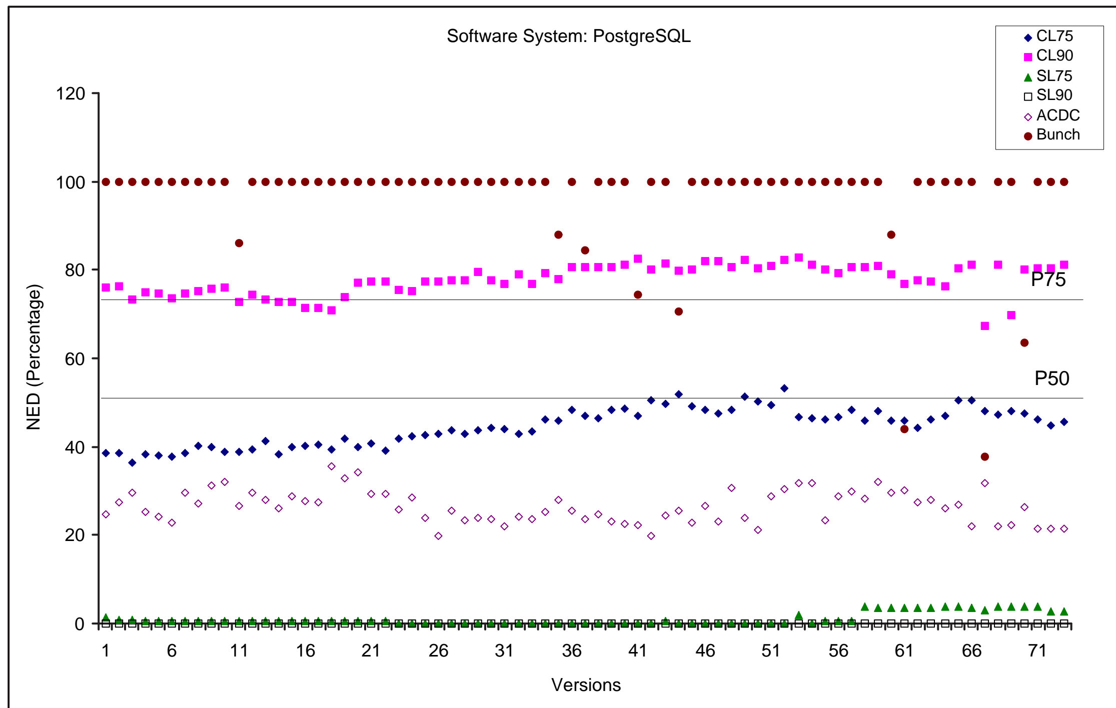


Figure 5.7: NED-based distribution comparison wrt PostgreSQL

where A_i ($1 \leqslant i \leqslant K$) denotes K algorithms and $NS(A_i, S)$ denotes the NED series obtained from the clusterings produced by A_i with regard to system S. The $P75$ and $P50$ are series comprising values 0.75 and 0.50 respectively. The $Score_{NED}$ tells whether a NS is above another NS. If so, it means that the algorithm producing the former NS has a better cluster distribution than the algorithm producing the latter NS. The HML_{NED} evaluates whether an algorithm really has a good cluster distribution or not. If A_i has a score of H, it means that 80% of the clusterings produced by A_i have a NED value greater than 0.75.

Ordinal evaluation

We conducted a similar ordinal evaluation as we did to stability and authoritativeness. For brevity, we only give the final NED-based ordering as follows, omitting the relative scores and HML scores.

Low	Medium	High
SL90 SL75 ACDC CL75	CL90	Bunch

5.6 Discussion

Table 5.5 provides a summary of ordinal evaluation on which our discussion is based.

Algorithm	Stability	Authoritativeness	Non-Extremity
CL75	Medium	Low	Low
CL90	Medium	Low	Medium
SL75	High	Low	Low
SL90	High	Low	Low
ACDC	High	Low	Low
Bunch	Low	Low	High

Table 5.5: A summary of ordinal evaluation of clustering algorithms

SL75 and SL90 are the most stable but the least useful

Figures 5.6(a) and 5.6(b) show that SL75 and SL90 have a tendency of merging smaller clusters (including singletons) one by one into one super large cluster. As the cut point is increased from 0.75 to 0.90, the number of clusters in the clustering decreases and the super

large cluster grows. The super large cluster functions like a black hole which eventually attracts every surrounding smaller cluster. The files in the black hole account for roughly 84.8-86.3% of all the files in the system in the case of SL75 and 79.8-98.7% in the case of SL90. This explains why SL75 and SL90 appear more stable than the other four algorithms. Once the majority of the files are put into one super large cluster, the clustering bears little similarity to the directory-based authoritative clustering. In terms of MoJo, the number of moves and joins is almost equal to the number of files because nearly every file in the super large cluster needs to be moved to a directory-based authoritative cluster and every singleton cluster needs to be joined. This explains why SL75 and SL90 are the least useful. A clustering of the majority of files into one black hole is equivalent to no clustering.

Algorithms in CL are less stable than those in SL

Because of the complete linkage update rule [AL99], the algorithms from the CL class are known to produce more compact clusters than those from the SL class. Raghavan and Yu have shown that graph theoretic clustering methods that produce more compact clusters are less stable [RY81]. The obtained stability ordering agrees with the theoretic results (see Section 5.5.1).

Bunch is the least stable with the best distribution

In our configuration of Bunch, every time a better neighbor is being sought after, 20% of the entire search space is randomized and at least 55% is examined in order to find a better MQ (Model Quality) [MMCG99]. As a consequence, the hill climbing algorithm in Bunch does not perform in a deterministic way and it even produces different decompositions for the same input graph. By contrast, the other algorithms use less randomization.

Bunch (including NAHC and SAHC) produces the best cluster distribution. It favors

neither a group of very small clusters nor an overwhelmingly large cluster. This can be seen from Figure 5.6(e).

ACDC does not gain from overusing program patterns

Since ACDC is a program pattern based algorithm, we had expected that it would produce clusterings more similar to the authoritative clusterings we obtained. However, it did not meet our expectations.

ACDC internally has an upper limit for the cluster size which is set to 20 by default. However, Figure 5.6(f) shows that ACDC produces both very large and very small clusters. To investigate what went wrong in ACDC, we manually examined the clusterings obtained from versions 12, 24, 36, 48, 60, and 72 of PostgreSQL. We found two problems related to the use of program patterns in ACDC.

- The body-header pattern [TH00a] results in the grouping of the interface file (`.h`) and its implementation file (`.c`). However, clustering of the obtained body-header pairs is not done sufficiently. This results in a relatively large number of small clusters of size 2.

- The subgraph dominator [TH00a] is overused. For example, the file `utility.c` located in the directory `tcop` was recognized by ACDC as a dominator. After we removed `utility.c` from the input graph, ACDC cut the size of the largest cluster approximately by 50%. After we further removed 2 or 3 dominators from the input graph, no cluster in the obtained clustering contains more than 100 source files. We suspect that ACDC has internal defects in processing subgraph dominators.

5.7 Related Work

Current research is mainly focused on designing advanced algorithms which partition large software systems into meaningful subsystems. For example, early work by Belady *et al.* identified automatic clustering as a means to produce high-level views of large software systems [BE81]. Bunch has evolved to become a suite of algorithms to fit into various contexts in reverse engineering [DMM98, MMCG99]. In addition, a variety of evaluation frameworks have been proposed to evaluate the quality of clustering techniques [KE00, LG97]. However, current clustering techniques have not been sufficiently tested and evaluated against lifetime versions of systems from diverse domains. In particular, the stability of clustering algorithms has not attracted much attention.

Tzerpos and Holt examined the stability of a number of clustering algorithms by means of generating randomly perturbed versions of an example system and measuring differences between the obtained clusters and the one obtained from the original version of the system [TH00b]. We argue that random perturbation of a fixed size system is insufficient in simulating how changes occur in a real world software system, since changes to software systems rarely occur in a random fashion and most software systems are continuously growing in size. To be faithful to the reality, we conducted stability comparison on evolutionary data extracted from several target systems. This work shows that stability comparison of clustering algorithms should be augmented with the analysis of meaningfulness (adherence to authority and non-extreme distribution). Otherwise, stability comparison could be misleading (*i.e.*, seeing only one side of a coin).

5.8 Conclusions

We have asked the question of how useful clustering algorithms might be in large software projects undergoing continual change and growth. We investigated six automated clustering algorithms which represent a range of clustering techniques and which are supported by available batch implementations. We selected five open source software systems, each having approximately 70 monthly versions. These systems represent a range of applications. So they can be expected to be a reasonable testing base for clustering algorithms.

We proposed three criteria to evaluate the usefulness of software clustering algorithms and ran experiments to measure how well each clustering algorithm satisfies these criteria. Table 5.5 provides a brief view of the overall results indicating that:

- On the stability criteria (Do successive versions of a system have similar clusterings?), three algorithms (SL75, SL90 and ACDC) are ranked as having high quality, two (CL75 and CL90) as medium and the remaining Bunch as low.
- On the authoritativeness criteria (Do the clusterings reasonably resemble the implemented system structure?), all the six algorithms are ranked as having low quality.
- On the extremity criteria (Are non-extreme cluster distributions normally produced?), Bunch is ranked as high, CL90 as medium, then CL75, ACDC, CL75 and SL90 as low.

Although SL75 and SL90 score high on stability, it is largely due to the fact that they repeatedly produce overly large or many too small clusters, as indicated by their low NED ranking. It suggests that these two algorithms may not be that helpful in practice. The fact that all the six algorithms are ranked low on authoritativeness suggests that they may not be mature enough for use in production on large systems undergoing evolutionary change.

However, it is also possible that our technique for generating authoritative clusterings is biased toward the as-implemented structure of the target system.

These results are discouraging, suggesting that, for large systems such as we used as a basis for our testing and for the criteria we chosen, more work needs to be done before these clustering algorithms are ready to be widely adopted. However, it may be that such algorithms are useful in less stringent environments. For example, these algorithms may be useful in a reverse engineering exercise by producing a basic partitioning of a system to eliminate a significant amount of manual effort.

Our hope is that our results spur on further efforts both to create/improve automated clustering algorithms and to subject these algorithms to empirical evaluations such as we have reported in this chapter. As a consequence, new clustering techniques may eventually emerge to satisfy the three criteria and can be used to study the evolution of design through system lifetime.

Chapter 6

Fractal Nature of Software Evolution and SOC Dynamics

This chapter examines eleven open source software systems and shows empirical evidence that fractal structures are present in the change history of all these systems. In our analysis, fractal structures are identified and measured as power laws throughout the lifetime of a system. More specifically, (1) the probability distribution of software change sizes follows a power law; and (2) the time series of change exhibits long range correlations with power law behavior. The existence of such spatial (across the system) and temporal (over the system lifetime) power law behavior strongly suggests that *Self-Organized Criticality* (SOC) [BTW87] occurs in the evolution of open source systems. As a consequence, SOC may be established as a useful conceptual framework for understanding open source software evolution dynamics (*i.e.*, the mechanism and cause of change and growth). We provide a qualitative explanation of software evolution based on SOC. We also discuss some potential implications of SOC to software practices and evolution.

6.1 Introduction

The laws of software evolution formulated by Lehman [Leh97] represent a major intellectual contribution to understanding software evolution dynamics (an underlying cause of change or growth). Lehman's eight laws are empirically grounded on observing how closed source industrial software systems such as IBM OS/360 were developed and maintained within a single company using conventional management techniques [LRW+97]. As summarized in Table 1.1, the laws suggest that software systems must be continually changed in response to external forces such as new functional requirements and hardware upgrade.

Outside the field of software evolution, a wealth of knowledge has been gained in understanding the change behavior of *complex systems* as diverse as sandpiles [BTW87], power blackouts [CNDP04], earthquakes [Sch90] and even biological evolution [SMBB97]. These systems are complex in the sense that no single characteristic size can control their changes (responses) over time. In other words, changes of any size can occur. For example, a power blackout may strike a street, a city, a state and all the way up to a country. The simplifying aspect of these systems is that their statistical properties can be measured as power laws in space and in time. In 1987, Bak, Tang and Wiesenfeld proposed *Self-Organized Criticality* (SOC) to explain such typical power law behavior [BTW87]. SOC was set out as an ambitious effort for explaining the existence of ubiquitous fractal structures in nature. It has two important signatures: (1) power law distribution of dynamical responses, and (2) long range correlations with power law behavior in time series of response [CNDP04, SMBB97]. In Section 6.2, we will provide a brief introduction to fractals, power laws and long range correlations and explain how these concepts are related to SOC.

Do software systems follow the SOC dynamics during their evolution? If so, the knowledge gained from studying complex systems can be used to enhance our understanding of software evolution. The above question arises from several previous studies which include:

- Software evolution as punctuated equilibrium

  In Chapter 4 we examined the structural evolution of software systems at the implementation level. We observed that three open source systems (OpenSSH, PostgreSQL and Linux) evolved through an alternation between long periods of small changes and short periods of large avalanche changes [WSHH04].

- Biological evolution as a self-organized critical phenomenon

  Recent studies have shown that fluctuations in fossil record time series are self-similar, exhibiting long range correlations with power law behavior [SMBB97]. The existence of such fractal structures means that, when examining a given time frame, some basic properties (*e.g.*, mean and standard variance) remain the same as those obtained from the entire time series if a change of scale is performed. SOC was suggested as a useful way of understanding how long periods of small extinctions are interrupted by mass extinctions [BS93, SMBB97]. The structural evolution of a software system exhibits similar characteristics of punctuation as fossil record, suggesting that SOC may also be useful for explaining software evolution.

- Open source movement as a self-organized collaborative network

  In comparison to traditional industrial systems, open source systems are largely developed based on a less strict control and management model [Ope05a, Ray99]. Spontaneous collaboration is promoted and backed by decentralized developer communities across the Internet [Ray99]. Researchers such as Madey and Koch suggest that open source projects can be seen as a self-organizing phenomenon featuring self-selection of tasks, leadership and spontaneous collaboration among developers [MFT02, Koc04]. The main empirical evidence they present is the power law distribution of open source project sizes (the numbers of developers) and the power law distribution of developer

contributions (the number of commits to the source control repository).

Based on the promising results from prior research, we feel that SOC may be established as a useful conceptual framework for describing and explaining software evolution. In this chapter we describe our effort in seeking evidence for SOC in open source software systems and discuss the implications of SOC to software practices.

The remainder of this chapter is organized as follows. Section 6.2 introduces basic terms and concepts used in this chapter. Section 6.3 explains empirical data collection with regard to software changes and time series of change. Section 6.4 analyzes the existence of power laws in the evolution of open source systems. Section 6.5 provides a qualitative explanation of software evolution based on SOC. We also discuss the potential implications of SOC to software practices. Section 6.6 discusses some threats to the validity of our work. Section 6.7 considers related work and Section 6.8 concludes this chapter.

6.2 Background

This section provides a brief introduction to fractal, power law, R/S time series analysis and SOC. Readers who are familiar with these concepts can skip to the next section.

6.2.1 Fractal

Fractals are mathematical or natural objects that are made of parts similar to the whole in some way. A fractal object is scale invariant since it has a self-similar structure that occurs at different scales. Many natural phenomena are self-similar such as trees and shorelines. A small branch of a tree looks like the whole tree due to the existence of branching structures. When the length of a shoreline is measured using the box counting method, the length of any segment can cover the same number of mesh boxes as the whole shoreline if a change

of scale is performed. Mandelbrot provides a detailed explanation of fractals in his famous book *The Fractal Geometry of Nature* [Man82].

6.2.2 Power Law

A power law relationship between two scalars x and y can be written as follows:

$$y = C \cdot x^k$$

where C is the constant of proportionality and k is the exponent of the power law. Such a power law relationship shows as a straight line on a log-log plot since, taking logs of both sides, the above equation is equal to

$$log(y) = k \cdot log(x) + log(C)$$

which has the same form as the equation for a straight line

$$Y = k \cdot X + c$$

The power law $f(x) = C \cdot x^k$ has a property that relative change $f(sx)/f(x) = s^k$ is independent of x. In this sense, $f(x)$ lacks a characteristic scale or is scale invariant. Consequently $f(x)$ can be related to fractals (sometime even called fractals) because of its scale invariance.

Power Law Distribution

Our study is concerned with a special kind of distribution called power law distribution in which the Probability Density Function (PDF) of size s is specified as $P(s) \sim s^{-\alpha}$ and the tail Cumulative Distribution Function (CDF) of sizes greater than or equal to s is specified as $D(s) \sim s^{-\beta}$. The relationship between α and β is $\beta = \alpha - 1$ [New05]. Since β can be

conveniently estimated using simple linear regression on log-log plots without binning data points, we choose to estimate β rather than α in this study.

Power laws are observed in many fields including physics, biology, geography, sociology and economics [New05]. For example, the distribution of earthquakes is found to follow a power law behavior $P(E) \sim E^{-\alpha}$ where E is the amount of energy. The exponent α exhibits some geographical dependence and is found to be in the interval from approximately 1.8 to 2.2 [Jen98]. The distributions of firm sizes [Axt01, GGP03] and of open source project sizes [HJ02, Koc04] are also found to follow power laws.

6.2.3 Time Series Analysis

Time series data is commonly used to characterize the evolution of software systems. For example, Lehman *et al.* studied the evolution of IBM's operating system OS/360 by means of observing system growth measured in terms of the number of source modules and number of modules changed for each release [BL76, LB85]. Turski performed a regression analysis of results from these case studies and proposed the inverse-square model. It suggests that system growth is inversely proportional to system complexity and that system complexity is proportional to the square of system size [Tur02]. Such regression analysis of time series is useful for understanding the nature of software evolution. In this study, we are however interested in studying the fractal properties of time series recovered from the change history of a software system.

A widely used statistical analysis technique for time series is Rescaled Range Analysis which is also referred to as the R/S statistic. It was formulated by Hurst in 1951 [Hur51]. Hurst is an English hydrologist who worked on the Nile River Dam project in the early 20th century. He observed that floods of the Nile River could be characterized as a persistent phenomenon, *i.e.*, heavier floods were accompanied by above average flood occurrences

and minor floods were followed by below average occurrences. Based on this observation, Hurst invented the R/S statistic which calculates the power law relationship between the rescaled adjusted range $R/S(\tau)$ and the time lag τ:

$$R/S(\tau) \sim \tau^H$$

where H is known as the Hurst exponent. The rescaled adjusted range $R/S(\tau)$ is defined as the mean of $R(\tau)/S(\tau)$ over m blocks of τ successive data points and measures how fast the range of the blocks grows as τ increases. The definition is as follows:

$$R/S(\tau) = \frac{1}{m}\sum_{i=1}^{m}\frac{R_i(\tau)}{S_i(\tau)}$$

where $R_i(\tau)$ and $S_i(\tau)$ are the self-adjusted range and the standard deviation obtained for the ith block of τ data points. If the τ data points in the ith block is re-numbered as $\{x_1, x_2, ..., x_\tau\}$, the calculation of $R_i(\tau)$ and $S_i(\tau)$ is as follows:

Standard deviation: $$S(\tau) = \sqrt{\frac{1}{\tau}\sum_{t=1}^{\tau}(x_t - \bar{x}_\tau)^2}$$

Self-adjusted range: $$R(\tau) = \max_{t=1}^{\tau} X(t,\tau) - \min_{t=1}^{\tau} X(t,\tau)$$

Cumulative deviation: $$X(t,\tau) = \sum_{u=1}^{t}(x_t - \bar{x}_\tau)$$

Mean: $$\bar{x}_\tau = \frac{1}{\tau}\sum_{t=1}^{\tau} x_t$$

The Hurst exponent H reflects the persistence of data in a time series. According to the value of H, natural or man-made temporal processes can be classified as follows:

- *Uncorrelated* if $H = 0.5$. A random walk is uncorrelated. Informally, one can think of that future events are not influenced by previous ones and also do not carry memory from the past.

- *Long term correlated* if $H > 0.5$. Processes from this category often have long runs of consecutive values above or below the mean. The Nile River has a value of $H = 0.91$ as calculated by Hurst [Hur51]. This value implies that flood occurrences of the Nile River are not purely random but somehow temporally dependent. This sometimes is referred to as *long range correlations*, *long range dependence* or *long memory effects*.

- *Long term anti-correlated* if $H < 0.5$. A process from this category produces anti-persistent time series in which a value above the mean is more likely to be followed by a value below the mean and vice versa. Such behavior is observed in mean-reverting processes such as interest rate change.

The Hurst exponent is directly related to the fractal dimension of a time series by the relation $D = 2 - H$ [Spr03]. The fractal dimension D measures the smoothness of fractal time series. A larger Hurst exponent leads to a smaller fractal dimension and a smoother surface. Different time series can be quantified by using the R/S analysis to estimate their associated Hurst exponents. Such quantification allows us to recognize similarities between different temporal processes (including software evolution), thereby eventually leading to recognizing underlying unifications that might otherwise have gone unnoticed.

Long range correlations in a time series can also be analyzed using the popular power spectral analysis based on Fourier transformation. If fractal structures are present, a power spectral density function in the form of $S(f) \sim f^{-\alpha}$ with $\alpha > 0$ can be obtained in low frequencies where $S(f)$ falls as a power law[1] [Ber94]. The Hurst exponent is closely related to the power spectral exponent α by the relation $\alpha = 2H + 1$ [Spr03]. For simplicity, we restrict our analysis to the R/S statistic.

[1] The α exponent of $S(f) \sim f^{-\alpha}$ is not the same as the α exponent of power law distribution $P(s) \sim s^{-\alpha}$.

6.2.4 Self-Organized Criticality

In 1987 Bak, Tang and Wiesenfeld proposed Self-Organized Criticality (SOC) in order to explain the widespread occurrences of spatial fractals and fractal time series (also known as $1/f$ noise or flicker noise) in nature [BTW87]. According to this theory, complex systems which consist of interacting components can exhibit some general characteristic behavior in time ($1/f$ noise) as well as in space (self-similar fractals) spontaneously.

The $1/f$ noise is a ubiquitous phenomenon [Mil02]. For a system, a time series (signal) is obtained by measuring some of its time-dependent quantities. If the power spectrum of the obtained signal behaves like $f^{-\alpha}$ with $\alpha \approx 1$, the system is said to exhibit $1/f$ noise. Self-similar fractals are another widely observed natural phenomenon including snowflakes, mountain landscapes and shorelines [Man82]. The $1/f$ noise and self-similar fractals are the most important diagnostics of SOC and are commonly measured as power laws [BTW88]. If a system exhibits power laws without apparent tuning then it is said to follow the SOC dynamics.

6.3 Data Collection

6.3.1 Software Change

In this study, we define a software change as a set of source files which are modified together for some reason. For example, a change can contain a number of source files modified by a developer to remove a defect. We consider two kinds of change: (1) changes recovered from source control repositories (*e.g.*, CVS) and (2) structural changes obtained by contrasting subsequent snapshots (*e.g.*, daily builds). For C and C++ programs, we consider files with the following extensions: `.c`, `.C`, `.cc`, `.cpp`, `cxx`, `.c++`, `.h`, `.H`, `.hh`, `.hpp`, `.hxx` and `.h++`.

The size of a change is measured as the number of files contained in that change.

CVS Change

A CVS change, which is recovered from the CVS repository, contains files committed by the same developer with the same log message and at the same time. The term "same time" in this context means that the files are committed in a short period. Zimmermann *et al.* describe a number of methods for recovering changes from CVS repositories [ZW04]. One of these methods is called the sliding time window protocol, which relies on a maximal time gap to determine whether two subsequent checkins belong to one change. The change log tool *cvs2cl* uses such a protocol to recover changes from the CVS repository automatically [FOP02]. We use *cvs2cl* in this study.

The changes recovered from a source code control repository may be related to different types of task such as bug fix, feature modification, functional improvement and refactoring. However, we do not differentiate these tasks in our change recovery.

Structural Change

A structural change contains files which satisfy the following requirements: (1) they have outgoing dependencies added or deleted; and (2) they are connected in an isolated subgraph within a delta graph obtained by contrasting two subsequent snapshot versions. The delta graph shown in Figure 6.1 contains five files B, C, D, F, G and H as well as six added and deleted dependencies among these files. The files B, C, F and H form an isolated subgraph in which B, C and H have changed their outgoing dependencies but F does not. According to the definition, B, C and H form a structural change with F excluded. Similarly, D forms a structural change by itself.

A structural change can be a subset of a CVS change or may span multiple CVS changes

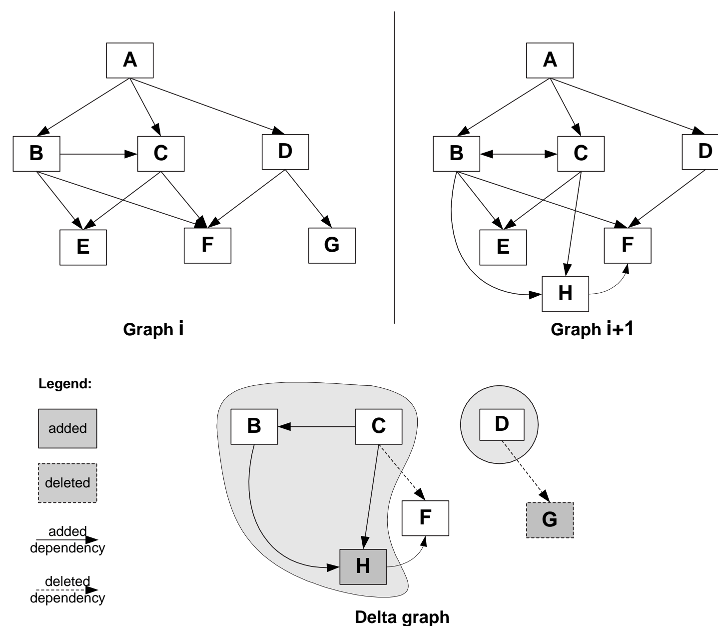


Figure 6.1: A delta graph between two adjacent versions

depending on the snapshot extraction frequency. Comparing two snapshots separated only by a single CVS change may produce structural changes smaller than that CVS change. For a long lived software system with tens of thousands of CVS changes, it can be difficult to obtain structural changes through system lifetime. If we conduct a comparison of two snapshot versions over an extended period of time (*e.g.*, one day), obtaining all structural changes will be relatively easy to do. In such a case, a structural change may span multiple CVS changes.

Our open source evolution database (as described in Chapter 2) provides a large number of software system models. Such a database allows us to compare subsequent snapshots or releases throughout a system's lifetime. In this study, we collect structural changes either on a daily basis or on the basis of releases if daily snapshots are not available.

6.3.2 Time Series of Change

A time series can be used to record change fluctuations throughout the lifetime of a software system. In this study, we measure the amount of change on a per-period basis by summing up the sizes of changes in each period. The time series of a software system can be obtained on a daily basis or via comparing consecutive releases if daily snapshots are not available. We are interested in studying the existence of long range correlations in time series because it represents the temporal signature of SOC. The R/S analysis is a powerful mathematical tool we can use to compute such a signature.

6.4 Examining Fractals in Software Evolution

This section examines empirical evidence for fractal properties found in the change history of open source software systems. The following phenomena are our main concern.

- Power law distribution of software changes
- Long range correlations in time series of change

For brevity, we describe the results obtained from GCC (GNU Compiler Collection) [GCC02] in detail and then summarize the results obtained from ten more software systems, which include FreeBSD, OpenBSD, NetBSD, PostgreSQL, PHP, KSDK, KOffice, OpenSSL, Ruby and Linux. A brief introduction to these systems is given in Appendix A.

6.4.1 Power Law Distribution of Software Changes

In our analysis of distributions of changes in GCC, the quantity being plotted is $D(s)$. We define $D(s)$ as the tail Cumulative Distribution Function of change sizes.

Power Law Distribution of CVS Changes in GCC

Fig. 6.2(a) displays a log-log plot, which was obtained from a total of 40,034 CVS changes recovered from the CVS repository of GCC. The recovered changes cover more than 8 years (1997/08/11–2005/09/09). This plot follows an approximately straight line. An ordinary least squares (OLS) linear estimation on the logarithmic scale of base 10 gives the following:

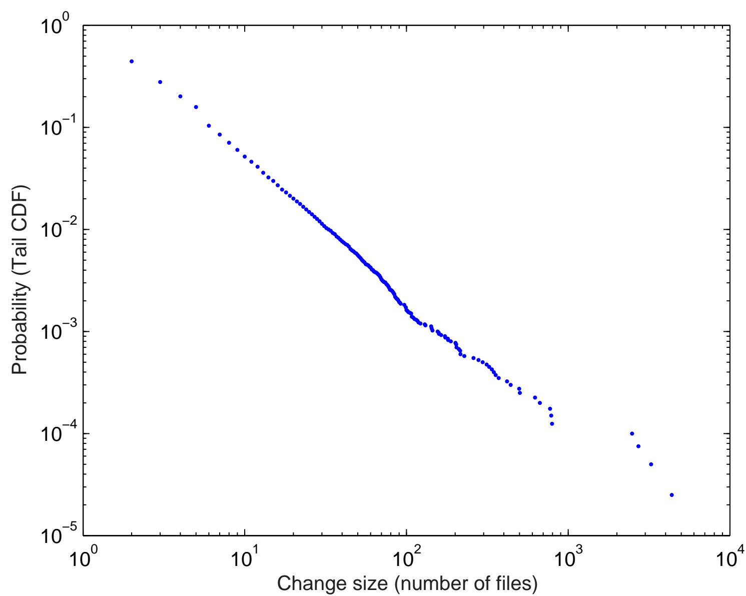
$$D(s) \sim s^{-\beta}, \ \beta = 1.3237$$

The function $D(s)$ has a property that relative change $D(ks)/D(s) = k^{-\beta}$ is independent of size s. $D(s)$ is scale invariant and hence can be seen as a statistically self-similar object. In GCC, large changes occur less frequently than small ones. But how much rarer are large changes? We can estimate the answer using the obtained $D(s)$ equation. The probability of making a change that involves more than 100 source files is extremely low, roughly less than 0.16%.

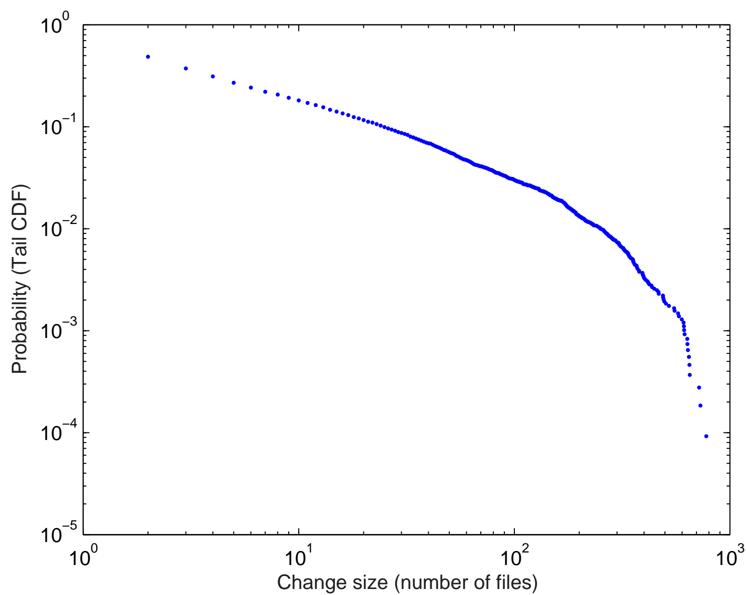
We have also found that similar power law distributions hold for individual years from 1998 to 2005 in GCC. The obtained scaling exponents vary from 1.29 to 1.34. This suggests that yearly distributions share a similar scaling behavior with each other and also with the lifetime distribution.

Power Law Distribution of Structural Changes in GCC

After observing that the size distribution of CVS changes followed a power law relationship, we were curious to know whether other kinds of software change have a similar distribution. We examined the size distribution of daily structural changes over the same development period of GCC. A power law was observed with all changes consisting of more than 100 files neglected. It is shown as a log-log plot in Fig. 6.2(b). An OLS linear fit on the logarithmic



(a) Size distribution of CVS changes ($\beta = 1.3237$)



(b) Size distribution of structural changes ($\beta = 0.7482$)

Figure 6.2: Tail cumulative distribution of change sizes for GCC

scale gives the following:

$$D(s) \sim s^{-\beta},\ \beta = 0.7482$$

The exponent β is estimated with $1 \leqslant s \leqslant 100$. It is interesting that this distribution begins to deviate from the power law roughly for $s > 100$. This suggests that a massive structural change involving more than 100 source files is extremely rare and not governed by the power law. A large change may structurally depend on hundreds of other files in the system. This makes it difficult to change a substantial number of structural dependencies among several hundred files.

The long tail deviation from power law may be caused by the finite-size effect [BTW88]. We suspect that the deviation would appear at larger change sizes if the number of files in GCC grows significantly. We have empirically observed similar deviations around $s = 100$ in several other open source software systems including KSDK, KOffice, OpenSSL, PHP, PostgreSQL and Ruby. Compared to GCC, these systems have a smaller or roughly similar size. For Linux which is many times larger than GCC, the deviation appears at sizes around 350. These differences can be seen in data plots shown in Appendix B.

Power Distributions Observed in More Systems

We examined ten more open source systems and found they followed power laws to varying degrees. The obtained scaling exponents are summarized in Table 6.1. For brevity, the log-log plots of these systems are omitted from this chapter but provided in Appendix B.

We did not analyze the distributions of structural changes for three BSD variants since each of them is actually composed of a large number of smaller applications and libraries. It is beyond the scope of this thesis to study the evolution of structural dependencies among collaborative applications. Unlike structural changes, CVS changes are mostly within the boundary of each smaller application or library.

System	Period	CVS Change (CC)			Structural change (SC)		
		#CC	β	R^2	#SC	β	R^2
NetBSD	1993/03/20–2005/08/17	86,280	1.3072	0.9952	–	–	–
FreeBSD	1993/06/06–2005/08/17	72,021	1.3435	0.9916	–	–	–
OpenBSD	1995/10/18–2005/08/17	47,969	1.1796	0.9963	–	–	–
Linux*	1994/03/13–2005/07/15	–	–	–	5,042	0.3420	0.9902
PostgreSQL	1996/07/09–2005/09/09	10,797	1.2866	0.9907	3,140	0.8573	0.9929
GCC	1997/08/11–2005/09/09	40,034	1.3237	0.9853	10,835	0.7482	0.9915
KSDK	1999/01/01–2004/09/15	4,012	1.4305	0.9851	1,112	0.7096	0.9655
KOffice	1999/01/01–2004/09/15	2,2948	1.4326	0.9899	19,913	0.5634	0.9624
OpenSSL	1999/01/01–2005/07/16	3,934	1.2989	0.9912	872	0.8208	0.9815
PHP	1999/04/07–2005/09/09	15,558	1.2749	0.9882	2,198	0.7701	0.9822
Ruby	1999/08/13–2005/09/09	3,655	1.5022	0.9935	443	0.9101	0.9227

*: The structural changes of Linux are obtained by means of comparing consecutive releases over time.

Table 6.1: Scaling exponents for distributions of software changes

For Linux, which does not have a source control repository (*e.g.*, CVS) for public access, we were not able to analyze change log information or perform daily structural comparisons. We instead studied the structural evolution of Linux through comparing 524 public releases from 1.0 to 2.6.12.3. These releases are ordered strictly according to release dates in order to form a historical sequence. Both development and stable releases of Linux were included in the sequence.

Here are some observations.

- All the studied software systems have a power law distribution with regard to either CVS or structural changes. The quality of fit (R^2) values shown in Table 6.1 indicate

a strong linear relationship between function $D(s)$ and the number of source files on a logarithmic scale.

- The distribution of CVS changes has a larger scaling exponent than the distribution of structural changes. This observation is limited to a certain threshold value, beyond which a large structural change can have a much lower probability when compared to a CVS change of the same size. This can be seen from Figure 6.2. The distribution in Fig. 6.2(b) deviates from the power law at larger sizes ($s > 100$) and drops toward zero probability. No structural change is found to involve more than 800 source files in GCC on a daily basis. This kind of deviation is apparently more common in the distribution of structural changes than in the distribution of CVS changes.

- The scaling exponent for OpenBSD is different from the scaling exponents for FreeBSD and NetBSD. This suggests that products from a product family may exhibit slightly different behaviors. This perhaps is because both FreeBSD and NetBSD have a longer development history and consequently have more CVS changes than OpenBSD. However, further examination is needed to understand what causes such differences in β from system to system.

- The distribution of structural changes for Linux was obtained by means of comparing subsequent releases rather than daily snapshots. The time interval between any two adjacent releases varies between 5 days and 37 days except that release 2.3.99-pre9 is approximately 6 months away from release 2.4.0. Such a sampling frequency tends to favor large structural changes rather than small ones. As a result, Linux has the smallest scaling exponent.

- The distribution of structural changes for KOffice (see Fig. B.2(f)), when examined over the entire range of change sizes (roughly from 1 to 400), does not follow a power

law. This perhaps is because that KOffice is largely implemented in C++ and that CTSX we used to extract program structural dependencies is still limited in handling C++ source programs (see Chapter 2). Therefore, structural changes in KOffice may not be identified appropriately.

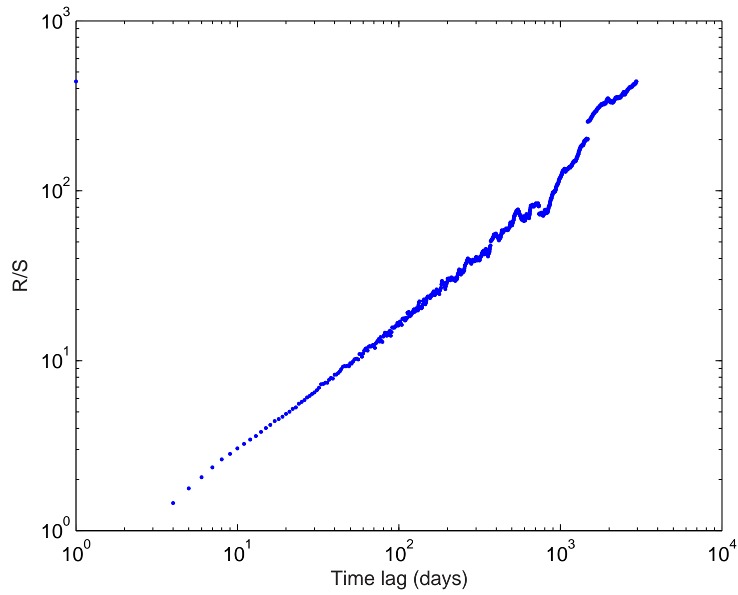
6.4.2 Long Range Correlations in Time Series

The essential nature of software evolution is change occurring spatially (across the system) and temporally (over time). Our observation of scale invariance in the size distribution of software changes leads to another interesting question. Does the change history of a software system exhibit self-similarity in the time dimension, *i.e.*, long range correlations with power law behavior? A positive answer to this question will suggest that SOC may be used to explain software evolution.

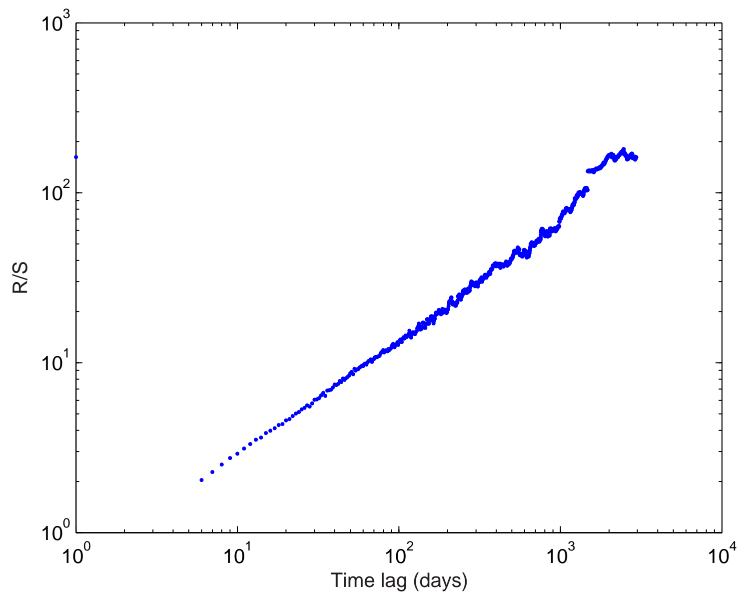
As pointed out in section 6.2.3, the R/S analysis can be used to analyze the presence of long range correlations in time series. Therefore, one way for answering the above question is to determine whether a time series of software change has a Hurst exponent greater than 0.5 (a characteristic value of random noise).

R/S Analysis of GCC

The R/S statistics plotted in Figure 6.3 have Hurst exponents with $H = 0.7711$ for time series of CVS change and $H = 0.6841$ for time series of structural change. These exponents are significantly above 0.5, thus indicating strong long range correlations. The results can be verified by means of randomly shuffling the original time series to eliminate correlations and re-applying R/S analysis. For GCC, a random shuffling always results in the reduction of H toward 0.5.



(a) R/S for time series of CVS change (H= 0.7711)



(b) R/S for time series of structural change (H= 0.6841)

Figure 6.3: R/S analysis of daily time series for GCC

System	CVS change			Structural change		
	TSC	H	R^2	TSC	H	R^2
NetBSD	98.6%	0.7340	0.9984	–	–	–
FreeBSD	96.4%	0.7586	0.9952	–	–	–
OpenBSD	96.5%	0.7181	0.9962	–	–	–
Linux*	–	–	–	96.2%	0.7491	0.9893
PostgreSQL	80.5%	0.7637	0.9969	52.1%	0.7029	0.9972
GCC	98.9%	0.7711	0.9973	84.8%	0.6841	0.9964
KSDK	53.5%	0.8096	0.9811	33.0%	0.7909	0.9872
KOffice	96.6%	0.8092	0.9921	90.3%	0.6967	0.9967
OpenSSL	56.6%	0.7354	0.9941	26.5%	0.7163	0.9894
PHP	94.7%	0.7545	0.9968	59.2%	0.6186	0.9948
Ruby	61.2%	0.6980	0.9936	14.7%	0.5129	0.9864

*: The time series of Linux is obtained by means of comparing consecutive releases over time.

Table 6.2: Hurst exponents from R/S analysis of daily time series

R/S Analysis of More Systems

The rescaled range analysis is applied to the same collection of software systems as analyzed in section 6.4.1. The obtained Hurst exponents are summarized in Table 6.2. Each Hurst exponent is estimated by considering time lags equal to 365 (days). But the largest time lag considered for Linux is 174 (releases) because its time series were obtained by comparing subsequent releases rather than daily snapshots. The time lag 174 accounts approximately for one third of 524 Linux releases we analyzed.

We define Time Series Coverage (TSC) as the ratio of the number of non-zero values to the total number of values in a time series. TSC measures how often changes occur over a

system's lifetime. In Table 6.2, Ruby has the lowest TSC (14.7%) with regard to structural change activities. This indicates that Ruby's structure was changed approximately once every seven days on average. Smaller systems such as KSDK, PHP and Ruby are less prone to change with regard either CVS or structural change activities. Larger systems such as GCC, Linux, KOffice and three BSD variants have a tendency of being changed every day. This is not surprising. For a larger software system, more developers are usually involved and changes occur more frequently.

The values of H obtained for all the time series of CVS change vary between 0.7 and 0.8. This suggests that these time series appear approximately equally correlated over the long term. In contrast, the values of H obtained from structural change have a wider span from 0.6 to 0.8 with Ruby excluded. The Hurst exponent of Ruby is 0.5129, a close indicator of random noise. This perhaps is because the time series of Ruby is too sparse (TSC = 14.7%) to yield any notable correlations.

6.4.3 Summary

We have presented evidence for the existence of fractal structures in the evolution of eleven open source systems. The observed fractal structures are measured as power laws in space (across the system) and in time (over system lifetime). Both CVS and structural changes follow power laws. However, CVS changes yield much stronger indications than structural changes.

6.5 Discussions

All the eleven systems we examined have evolved over many years in the real world. However, their change dynamics share a typical power law behavior which appears independent

of the details of every individual system. Where do these scale free dynamics come from? In this section we provide a qualitative explanation of software evolution dynamics from a perspective based on SOC.

Bak, Tang and Wiesenfeld used the sandpile model to illustrate the dynamics of a SOC system [BTW88]. Suppose a person starts to build a sandpile on a flat board by means of dropping grains of sand randomly, one grain at a time. The sandpile grows as many smaller piles are formed and their slopes increase continuously. The slopes at different locations will eventually reach a critical value; if more sand is added, sand slides will occur. As the sandpile is built up, the characteristic size of the largest sand slides (avalanches) grows until the state of criticality is reached, in which the size of the largest avalanche is equal to the size of the entire board. The dynamical behavior of the sandpile in criticality shows characteristics of $1/f$ noise and fractal structures. The quantity exhibiting $1/f$ noise is the sliding rate of sand measured over time and fractal structures appear in the form of power law distribution of sand slide sizes. Local random perturbations (sand drop) in criticality can result in responses (sand slide) of any size up to the entire system. The temporal and spatial power law behaviors are a direct consequence and extend over several decades on a macroscopic level.

The spatial and temporal power laws we have observed from the evolution of open source systems (see Section 6.4) suggest that software systems may follow the SOC dynamics. An analogy to the sandpile model can be drawn to explain the evolution dynamics of software systems in a qualitative way. Table 6.3 shows the proposed analogy which has four elements: *driving force*, *response*, *system state* and *relaxing force*. We now interpret the meanings of these elements.

Like a running sandpile, an evolving software system is continuously changed under the influence of various driving forces as diverse as customer requirements, hardware upgrade,

	Sandpile model	Software system
Driving force	sand drop	change request
Response	sand slide	change propagation
System state	gradient profile	release/iteration plan
Relaxing force	gravity	stakeholder demands

Table 6.3: Analogy between the sandpile model and software system

developer turnover, and development process and methodology. More specifically, changes are commonly made in response to concrete requests related to bugs, refactorings, features and etc. Such requests cause changes to propagate to different locations within the system. In this view, a change request is analogous to a sand drop and a change (CVS change or structural change) is mapped to a sand slide.

The system state and relaxing force are two complicated elements which need a careful explanation. For the sandpile, the system state is a matrix of maximum gradients covering all the locations, and the relaxing force is gravity which controls sand slides by reducing maximum gradients at appropriate locations if more sand is added. From our point of view, stakeholder demands can be mapped to gravity and the release/iteration plan to the sand gradient profile. The demands of stakeholders (*e.g.*, developers, architects and customers) must be satisfied. This is consistent with the law of *Continuing Change* [Leh97] stating that changes must be continuously made to satisfy user requirements. Changes are estimated, planned and performed at varying levels of priority. The relaxing force can be either release plans in the long term or iteration plans in the short term, controlling how and when to deliver the next release or prepare a workable product for the next cycle of development. For example, if a refactoring is needed, a maintainer (stakeholder) performs the refactoring task in several steps. After the refactoring is done, the maintainer re-gains satisfaction with

the system.

Criticality and Self-Organization

Given the above informal explanation of software evolution dynamics, one may ask when an evolving software system enters criticality and what self-organization means in the context of software evolution.

We studied eleven open source systems by analyzing the change history of each system after the first release was out or a reasonable amount of source code (normally tens of files) was developed. Since then power laws can be observed in the distribution of change sizes and in the time series of change. A software system may enter criticality no later than its first release.

As for the second question, an understanding of self-organization cannot be achieved by examining software systems without taking into consideration development methodologies and developers. In open source software projects, developers collaborate with one another spontaneously for some common purpose and they have freedom to modify and redistribute the source code and to work on the source code of their interest. A central organization in the name of core group or steering committee [BP03] may exist and provide some guidelines or advice but they do not control or command what individual developers should modify. Such an organization plays an important role in anchoring a broader community of users and developers and nurturing leadership and collaboration across the community [Hig99]. Spontaneous collaboration activities among developers eventually result in the first delivery of the system and then sustain the future evolution of the system.

Self-organization should not be confused with Lehman's third law of software evolution, *Self Regulation* [Leh97]. Self regulation is a control notion suggesting that positive and negative feedback controls are constantly and pervasively exercised during system evolution.

By contrast, self-organization is a configurational notion indicating spontaneous developer collaboration which is neither entirely directed by a central organization nor prescribed by a published process guideline [JS03].

Change Propagation

If a software system follows the SOC dynamics, what predictive power can one get? Can one foresee the extent to which changes propagate through the system? According to SOC, a complex system evolves at the edge between chaos and order where no single characteristic event size can control the system evolution [Jen98]. For example, the electricity power grid is postulated to operate in such a narrow region where power blackouts can not be limited to a certain small size such as a street block or a city [CNDP04]. It is difficult to predict where a blackout can occur and how far the blackout can propagate.

As an evolving software system responds to the changing environment and requirements, changes of varying sizes occur at different locations within the system. As indicated by the power law distributions we observed in the evolution of open source systems, a modification may be up to any sizes. The occurrences of change covering a significant part of a system or even the entire system, though rare, appear unavoidable in open source systems. This suggests that it can be difficult to predict the propagation of change. Necessary measures should be adopted to facilitate communication and/or collaboration between developers to prepare the developer, team, and organization for unexpected large changes. For example, Collective code ownership [Nor03] offers an effective strategy for encouraging collaboration between developers.

Agile Software Development

Agile Software Development is a conceptual framework for undertaking software engineering projects with the help of lightweight methodologies such as Adaptive Software Development [Hig99] and Extreme Programming [Ext04]. Generally speaking, agile methodologies value

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation, and
- Responding to change over following a plan

The above core values are regarded as the canonical definition of agile software development and commonly referred to as the *Agile Manifesto* [Man01].

Highsmith has advocated adaptive software development as an alternate approach between Monumental and Accidental software development in today's turbulent e-commerce world [Hig99]. His idea is drawn from complex adaptive systems theories including SOC. He emphasizes adaptability, speed and collaboration as the key elements to the success of software project teams who develop and manage high-speed, high-change and high-uncertainty projects. Other agile methodologies share similar ideas.

Agile software development has grown in recognition of complex systems theories (*e.g.,* SOC) which value adaptation over prediction or optimization [Man01]. The agile manifesto was summarized by many agile software practitioners through their hands-on experience in developing and managing software projects over years. The fractal phenomena we observed in open source software evolution provide empirical evidence for the existence of SOC. This can lend a hand to agile software practitioners in justifying their methodologies.

6.6 Validity Threats and Limitations

There are several threats to the validity of our work.

- The systems we studied are successful, large open source systems. There exist large numbers of small open source projects which neither attract many developers nor are maintained actively over a long period of time. Therefore, the eleven systems in our study are not representative of small or failed open source projects.
- Whether closed source industrial systems exhibit similar power laws in space as well as in time is unknown and needs to be empirically checked through future work.
- The definition and identification of a software structural change is not as accurate as those of a CVS change. This is further complicated by the inaccuracy of the source code extractor (CTSX) we used to prepare the structural data. When being applied to extract C++ programs, CTSX often produces many false cross-references which can make the identification of structural change more difficult. This perhaps explains why KOffice (largely implemented in C++) appear not to follow power laws during its structural evolution (see Fig. B.2(f)).
- Though we presented empirical evidence for power laws, we need adapt some known SOC theoretical models [BS93, SMBB97] to simulate software evolution and to verify our empirical findings. Only when theoretical results are consistent with our empirical findings, we can assert that open source software systems follow the SOC dynamics during their evolution.

6.7 Related Work

Recent studies of software evolution have been directed to examine power law distributions related to open source projects hosted on the web site of SourceForge [Sou05]. The mission of SourceForge is to enrich the open source community by providing a centralized place for open source developers to control and manage open source software development.

Hunt and Johnson studied downloads of software projects at SourceForge and found that projects sizes (number of software downloads) follow a Pareto distribution [HJ02]. A Pareto distribution is in fact a rank-based power law distribution [New05]. Their findings show there are a small number of exceptionally popular software projects such as Linux, Apache and Mozilla while most SourceForge projects are less popular. They suggest that studying median projects instead of exceptionally popular projects may be useful for identifying "best practice" for open source software development.

Similar power law distributions about open source projects were reported by Madey *et al.* [MFT02]. In their study project sizes are measured as the number of developers. Their results indicate that most open source projects at SourceForge have only one developer and only a small percentage have a larger ongoing team. In addition, they modeled open source projects as a collaborative social network with developers as nodes and joint membership in projects as links between nodes. The clusters (development teams) in the social network are connected by linchpin nodes which are developers playing an important role in transferring ideas and technology between separate development teams. The presence of power laws in such open source networks are suggestive evidence that open source software development can be modeled as self-organizing collaboration social networks.

Koch studied individual programmers' contribution to open source projects at SourceForge and found power laws [Koc04]. The contribution of a programmer is measured using the number of commits made by the programmer or the number of projects the programmer

worked on. The observed power laws indicate that only a small minority of programmers are responsible for the major growth of open source projects and that the collocation of projects in a virtual hosting community such as SourceForge does not significantly increase co-participation.

These studies have treated the open source software community as an ecology in which individual programmers collaborate with one another and ideas are nurtured and projects are delivered. Both Madey and Koch have suggested that the overall open source software development is a self-organizing cooperative system in which spontaneous collaboration as well as leadership (*e.g.*, chief programmers) contribute to the success of many open source projects. In contrast, we examine power laws during the evolution of individual open source systems and suggest that individual systems follow the SOC dynamics.

The work of Gorshenev and Pis'mak on explaining software evolution dynamics using SOC perhaps is the most relevant to our work [GY03]. They observed that the distribution of added lines (or deleted lines) follows power laws in three open source systems which are Mozilla, FreeBSD and Emacs. In our work power laws are studied with regard to file level changes and observed not only in distribution of changes but also in time series. We also suggested a qualitative explanation of software evolution dynamics based on Bak's sandpile model.

It is interesting to note that power laws have been observed at three levels of abstraction in open source software development, the project level [HJ02, MFT02, Koc04], the file level (our work) and the level of lines of code [GY03]. These observed power laws in the open source ecology make it promising to explain the evolution of the community as well as the evolution of individual projects using SOC. A unified framework based on SOC may be constructed in the future, within which software evolution can be explained and successful practices may be identified.

6.8 Conclusion

In this chapter we studied the existence of fractal structures in the evolution of open source software systems. Changes and time series of change are recovered from the change history throughout system lifetime and analyzed using probability distribution and R/S analysis. Fractal structures are identified and measured as power laws spatially (across the system) and temporally (over the system lifetime). The findings are presented in the form of power law distribution of change sizes and long range correlations in time series of change. Such spatial and temporal fractal structures strongly indicate that open source software systems may follow the SOC dynamics during their evolution. Our empirical findings support the view that the open source community can be seen as a self-organizing collaborative network [Hig99, Koc04, MFT02].

Chapter 7

Conclusions and Future Work

It is anticipated that a theoretical advance in understanding software evolutionary processes would lead to significant improvement to the current practices in developing and managing software products. As open source software (OSS) gains popularity, researchers have begun to take advantage of large numbers of OSS projects to substantiate their understanding of software evolution. Following this trend, we undertook an empirical study of OSS evolution with an objective of understanding the mechanisms and causes of change in OSS systems.

The work presented in this thesis has two main focuses. First, we presented techniques and approaches to extract evolutionary structural information from software systems in a timely and cost-effective manner. Our attention was directed to a multipurpose systematic approach to extracting C/C++ programs. Second, we proposed new ways of understanding software evolution based on *Punctuated Equilibrium* and *Self-Organized Criticality* (SOC) respectively and conducted empirical validation on several OSS systems.

We designed and developed a suite of program extractors for the C and C++ programming language. This extractor suite is called CX, a shortened name for C/C++ eXtraction. It covers several individual steps in the build pipeline of C/C++ programs and handles

both source code and binary code, thus providing systematic support for program extraction. This suite currently has four extractors, CTSX, CPPX, LDX and BFX, which allow for tradeoff between accuracy, efficiency and robustness to varying degrees. To complement these extractors, we further proposed a set of heuristics to aid the linking of program models extracted from individual files into a combined system model of reasonable accuracy. Combining the proposed heuristics with a fast and robust program extractor (*e.g.*, CTSX) provides a cost-effective means to extract software system models from historical sequences of releases and snapshot versions.

We proposed to view software evolution as Punctuated Equilibrium. In response to the changing environment and requirements, a software system undergoes many small changes and occasionally large avalanche changes over its lifetime. When observed from an external perspective, large avalanche changes can be correlated with major disruptive events, which are often related to architectural adaptation and major functionality update.

To observe punctuated evolution, we applied the evolution spectrograph and automated clustering techniques to capture conspicuous structural changes at the implementation level and the design level respectively. Aided by the evolution spectrograph, we observed that a number of OSS systems exhibited the characteristics of punctuation during the evolution of the implemented system structure. Unfortunately, current automated clustering techniques were found to be incapable of recovering system designs consistently, thus leaving our goal of carrying out an automated examination of punctuated evolution in OSS on a large scale unfulfilled for the time being.

Instead of viewing that software evolution is driven mainly by external forces (*e.g.*, new functional requirements), we presented a new perspective on software evolution dynamics. From this perspective, a software system responds to external forces by following an inherent dynamic mechanism called SOC. The SOC dynamics can be characterized by two

statistical signatures: (1) the probability distribution of change sizes is a power law; and (2) the time series of change exhibits long range correlations with power law behavior. We analyzed the change history of several open source systems to verify the existence of power laws and we found evidence for power laws in change distributions and time series. We further provided a qualitative explanation of software evolution by drawing an analogy to the Sandpile SOC model and discussed several potential meanings of SOC to software practices and evolution. Our observation of power laws in OSS evolution provides empirical grounds for establishing SOC as a conceptual framework for seeking a simple and unified explanation of diverse evolutionary phenomena.

In conclusion, we developed techniques and approaches to support software evolutionary data collection and investigation on a large scale, and we proposed and empirically validated new ways for understanding OSS evolution (as punctuated equilibrium) and its underlying dynamics (based on SOC).

7.1 Thesis Contributions

This section highlights the main contributions of this thesis.

1. **An Extractor Suite for C/C++**

   The suite supports systematic program extraction by covering individual steps in the build process of C/C++ programs and handling source code and binary code. It also allows for tradeoff between task-specific extraction needs.

2. **Program Linking Heuristics**

   The proposed heuristics are aimed at extracting software system models of reasonable accuracy when system configuration information is difficult or too costly to obtain.

3. **EvolDB - A Software Evolution Database**

   EvolDB contains tens of thousands of software system models extracted from eleven OSS systems. EvolDB can provide an empirical basis for analyzing software evolution in many interesting ways.

4. **Software Evolution Spectrograph**

   The evolution spectrograph provides a simple and scalable means to graphically display how a group of entities change over an extended period of time. This technique is useful for distinguishing patterns from a sheer volume of historical data.

5. **Punctuated Software Evolution**

   Viewing software evolution as punctuated equilibrium offers a simple way of describing many evolutionary phenomena such as ripple effect and discontinuous evolution. In this view, a software system undergoes long periods of small changes interrupted occasionally by large avalanche changes. We have observed that several open source systems exhibited characteristics of punctuation during the evolution of their implemented system structure.

6. **Evidence for SOC in Software Evolution**

   We observed power laws in the probability distribution of software change sizes and in the time series of software change. This provides empirical evidence for SOC in the evolution of OSS systems. Within the conceptual framework of SOC, punctuated software evolution can be explained.

7. **A Framework for Software Clustering Evaluation**

   This framework is based on three criteria: stability, authoritativeness and extremity. Using this framework, we compared several automated clustering tools by applying

them on large evolutionary data sets. The obtained results have shown that current automated clustering can not be used to support the analysis of software evolution.

7.2 Future Work

Possible future work that stems from this thesis is considered below.

7.2.1 Preprocessor-Based Program Extraction

The build pipeline for C and C++ programs typically has three steps: preprocess, compile, and link. The CX suite we developed for the C and C++ programming language does not support fact extraction in the preprocessing step. No extractor in the suite can accurately extract dependencies between macros and other types of program entity such as functions. We plan to extend the current extractor suite by adapting the C preprocessor, *cpp* [GCC02], to output program preprocessing information.

7.2.2 Evolution Visualization on a Large Scale

In the analysis of software evolution, visualization techniques can be effective for providing a global picture and revealing interesting evolutionary patterns. For example, the evolution spectrograph [WHH04, HWH05] can cope with large quantities of historical data recovered from a software system in a simple and scalable way. Similar techniques such as Evolution Matrix [Lan01] and Percentage Bars [GJR99] have been shown useful for exploring software system evolution in different ways. These techniques, though preliminary, have highlighted the potential benefits of visualization in handling the sheer volume of historical information recovered from various repositories such as source control repositories and bug databases.

Many questions remain to be answered about how to scale up evolution visualization. In particular, we feel that supporting evolution visualization with flexible browsing, querying and scripting capabilities would help researchers explore the evolution of a software system more effectively. For example, the evolution spectrograph can be combined with hierarchy structure visualization techniques (*e.g.*, SHriMP Views [SBM+02]) to provide an interactive exploration environment. In this environment, the evolution spectrograph is used to reveal global trends and the structural visualizer is used to examine the detailed system structure. Querying and scripting capabilities can be added to support environment customization for specific user needs.

7.2.3 Evolution Analysis of Software Architecture

There has been a tacit assumption that studying the evolution of a software system at the architectural level can help locate architectural problems hindering the long term evolution of the system. It is not uncommon that many large systems do not have a well-documented architecture. This makes architecture recovery as well as architecture comparison a daunting task to do over a long sequence of historical releases. Finding an cost-effective approach to recovering system architectures automatically or semi-automatically would produce new opportunities for analyzing software evolution.

One of our previous studies (see Chapter 5) has shown that current automated clustering techniques are not capable of recovering system architectures consistently. But there may be a workaround solution if we apply semi-automated clustering and perform a reasonable amount of manual work. For a software system evolving in punctuated equilibrium, releases in periods of punctuation can be automatically clustered and manually refined, and releases in periods of equilibrium can be clustered using Orphan Adoption [**?**]. Orphans refer to source files that do not belong to any cluster in an existing clustering. Throughout periods

of equilibrium, the number of orphans to be adopted into available clusterings is expected to be small. The empirical evaluation of this proposed approach is needed.

7.2.4 Evolution Simulation Based on Known SOC Models

Power laws are a fingerprint of the dynamics of Self-Organized Criticality (SOC) [BTW87]. Several theoretical models have been proposed to simulate self-organized critical systems. These models include the Sandpile model [BTW88], the Bak-Snappen model on an evolving ecosystem [BS93], and Solé's network model of macroevolution [SMBB97]. They have been used to account for empirically observed power laws in time as well as in space. For example, Solé *et. al* used their network model to verify statistics observed in the fossil record, thus providing theoretical evidence for SOC in macroevolution.

It would be interesting to adapt one of the above models to simulate software evolution and verify power laws we observed in OSS systems. By doing so, we will be able to obtain theoretical evidence that SOC does occur in OSS evolution.

Appendix A

A List of Open Source Projects

The following is a list of open source software projects, which we have either mentioned or studied in the previous chapters. These projects are listed in ascending alphabetical order.

1. **Emacs** [Ema04] is an extensible, customizable, self-documenting real-time display editor. At its core is an interpreter for Emacs Lisp, a dialect of the Lisp programming language with extensions to support text editing. Emacs is popular with computer programmers and other technically-proficient computer users.

2. **Fist Person Shooter (FPS) Computer Games**

   - **Cube** [Cub03] is an open source multiplayer and singleplayer FPS game built on an entirely new landscape-style engine. The engine combines high precision dynamic occlusion culling with a flexible form of map/geometry editing in-game.

   - **Quake** [id 03] is a multiplayer and singleplayer FPS game, which was developed by id-Software in 1996. Quake was the first FPS game whose multiplayer could be played against many people on the Internet rather than with only three other

people on a local network. Quake was also revolutionary due to its utilization of three dimensional polygons for both scenes and players.

3. **GCC** [GCC02] stands for the GNU Compiler Collection. GCC includes a number of programming language compilers for C, C++, Objective C, Fortran, Java, and Ada, as well as libraries for these languages. GCC is the key component of the GNU tool chain and provides standard compiler support for free Unix-like operating systems.

4. **Gnumeric** [Gnu03] is a spreadsheet program which is part of the Gnome Office.

5. **InnoDB** [Inn04] is a transaction-safe storage engine which has commit, rollback, and crash recovery capabilities. InnoDB is included as standard in all current binaries distributed by MySQL.

6. **Instant Messaging (IM) Clients** support instant person to person communication over a network such as the Internet.

   - **CenterICQ** [Cen04] is a text mode menu- and window-driven IM client.
   - **Gaim** [Gai04] is a multi-platform instant messaging application that supports many commonly used instant messaging protocols including AIM, ICQ, Yahoo, MSN, IRC, and Napster.
   - **Kopete** [Kop04] is an instant messenger for the KDE environment. Like Gaim, Kopete is designed to be a flexible and extensible multi-protocol system suitable for personal and enterprise use.
   - **Miranda IM** [Mir04] is is a multi-protocol instant messenger client. Miranda is designed to be resource efficient and easy to use. It is built on a basic framework with functionality implemented entirely through plugins.

7. **KOffice** [KOf04] is an integrated office suite for the K Desktop Environment (KDE). KOffice is developed as part of the KDE Project and consists of 12 major applications: KWord, KChart, KSpread, KPresenter, Kivio, Karbon14, Krita, Kugar, KPlato, Kexi, KFormular and Filters that permit KOffice to interoperate with other popular office suites such as OpenOffice and Microsoft Office.

8. **KSDK** [KDE04] is a software development toolkit designed for developers who work on the KDE project. KSDK offers a collection of tools for developing and debugging various kinds of KDE applications.

9. **KSpread** [KOf04] is a scriptable spreadsheet application which provides both table-oriented sheets and support for complex mathematical formulas and statistics. KSpread is part of the KOffice project.

10. **Linux** [Lin04] is a clone of the operating system Unix, written from scratch by Linus Torvalds and subsequently worked on by hundreds of developers who are loosely connected through the Internet. It aims towards POSIX and Single UNIX Specification compliance. The first official version of Linux, 1.0, was released in March 1994.

11. **Mozilla** [Moz04] is a free cross-platform internet application suite whose components include a web browser, an e-mail and newsgroup client, an HTML editor, and an IRC client. Mozilla also stands for an application framework, which comprises a collection of cross-platform software components such as layout engine, user interface toolkit, and support for web services. The Mozilla application suite and framework was based on the source code of Netscape Communicator released by Netscape Communications Corporation under an open source license in March 1998.

12. **MySQL** [MyS04] is a popular SQL-compliant relational database management system (DBMS). MySQL is available as free software under the GNU General Public License (GPL) and also under traditional proprietary licensing arrangements for cases where the intended use of MySQL is incompatible with GPL.

13. **OpenSSH** [Ope04a] is an open source implementation of Secure Shell (SSH) protocol suite of network connectivity tools. OpenSSH encrypts network communication traffic (including passwords) to effectively eliminate eavesdropping, connection hijacking, and other network-level attacks. Additionally, OpenSSH provides a myriad of secure tunneling capabilities, as well as a variety of authentication methods.

14. **OpenSSL** [Ope04b] is a cryptography toolkit implementing the Secure Sockets Layer (SSL v2.0/v3.0) and Transport Layer Security (TLS v1.0) protocols as well as a full-strength general purpose cryptography library. The OpenSSL project is managed by a worldwide community of volunteers that use the Internet to communicate, plan, and develop the OpenSSL toolkit and its related documentation.

15. **PostgreSQL** [Pos03] is a large, SQL-compliant object-relational database management system (DBMS). It first originated at the University of California at Berkeley and now has more than 15 years of active development with a globally distributed development team. PostgreSQL runs on all major operating systems, including Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64), and Windows.

16. **PHP** [PHP04], short for "Hypertext Preprocessor", is a reflective programming language used mainly for developing server-side applications and dynamic web content. PHP allows interaction with a large number of relational database management systems such as MySQL, PostgreSQL, Oracle, IBM DB2, and Microsoft SQL Server.

17. **OpenOffice** [Ope05b] is an office suite intended to be compatible with and compete with Microsoft Office. OpenOffice includes a word processor, spreadsheet, presentation, vector drawing, and database components.

18. **Ruby** [Rub04] is an interpreted scripting language for quick and easy object-oriented programming. It has many convenient features for text file processing and system management.

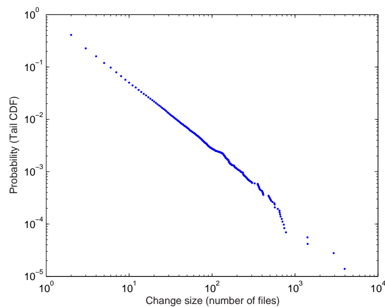
19. **Web Browsers**

   - **Dillo** [Dil04] is a web browser targeted at embedded platforms with a focus on fast page rendering and low memory usage.
   - **Epiphany** [Epi04] is the official browser for the Gnome Desktop. Epiphany is built on the Mozilla application framework.
   - **Konqueror** [Kon04] is the official web browser for the K Desktop Environment. It also supports general-purpose file browsing and management.
   - **Lynx** [Lyn04] is a text-only web browser for use on cursor-addressable, character cell terminals.
   - **Safari** [Saf04] is a web browser developed by Apple Computer for its Mac OS X operating system. Safari is built on the KHTML rendering engine from KDE.

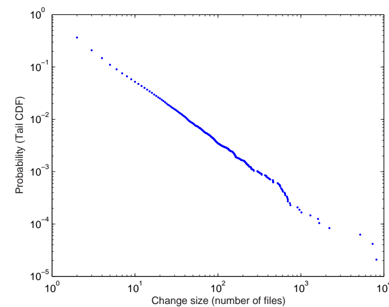
Appendix B

Distributions of Software Changes

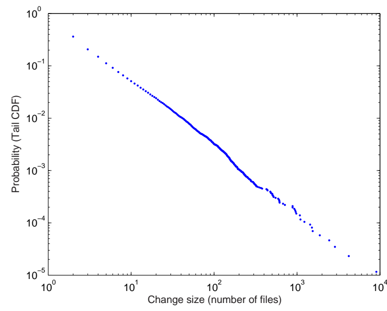
This appendix displays a number of plots we obtained from open source software systems. In each plot, the tail Cumulative Distribution Function (CDF) is plotted against the size of changes on the logarithmic scale of base 10. Two types of software change are considered. Figure B.1 shows distributions of changes recovered from the CVS source control repository. Figure B.2 shows distributions of changes obtained via daily system structural comparison. System structural changes were only obtained for the Linux kernel by comparing subsequent official releases since we could not get access to the source control repository of Linux.



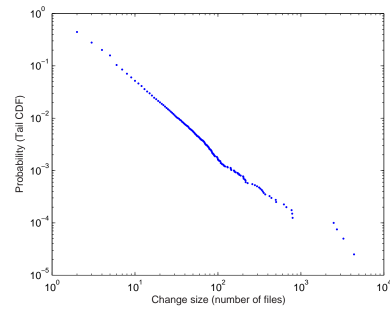
(a) FreeBSD



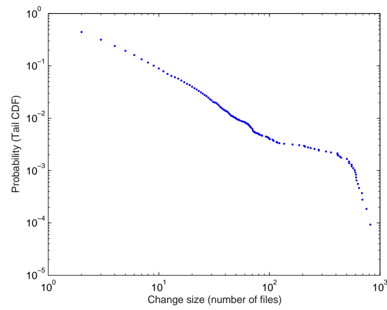
(b) OpenBSD



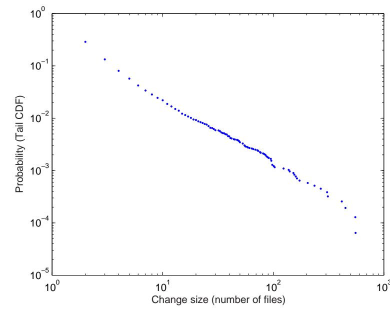
(c) NetBSD



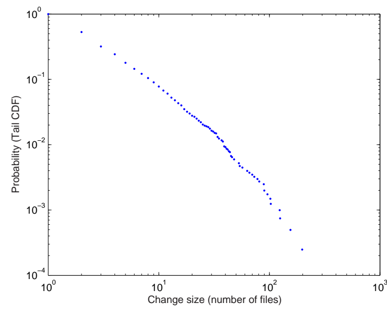
(d) GCC



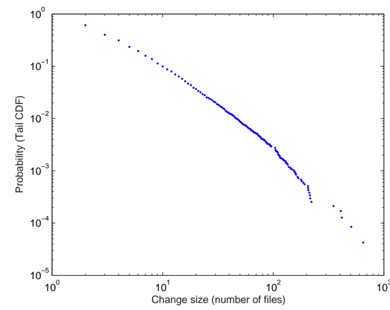
(e) PostgreSQL



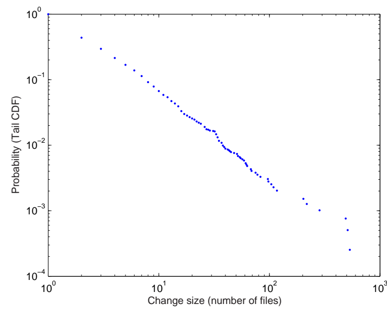
(f) PHP



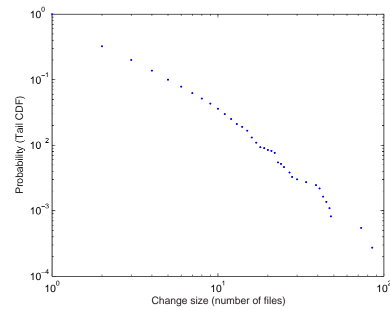
(g) KSDK



(h) KOffice

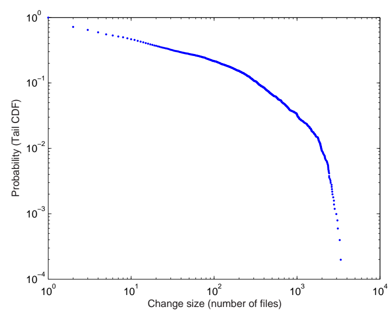


(i) OpenSSL

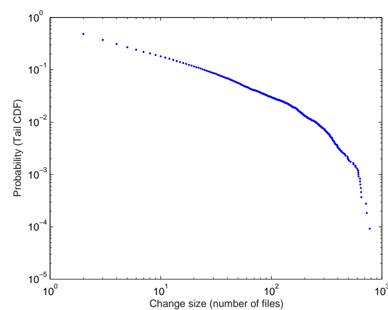


(j) Ruby

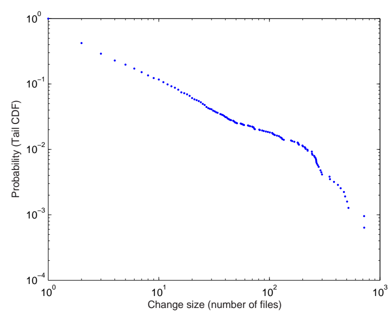
Figure B.1: Tail CDF of changes recovered from the CVS repository



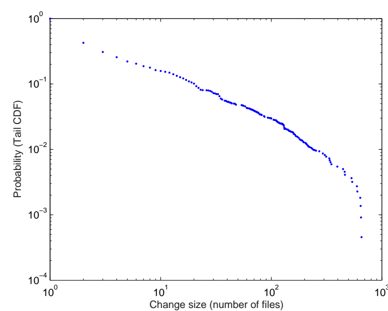
(a) Linux Kernel



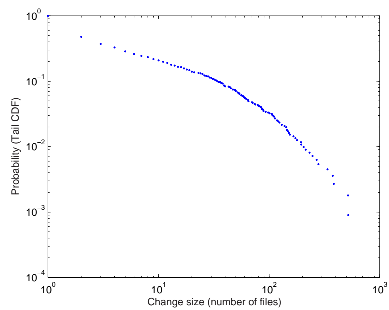
(b) GCC



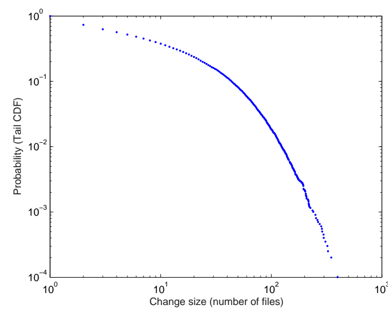
(c) PostgreSQL



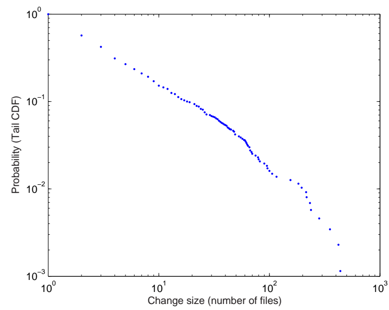
(d) PHP



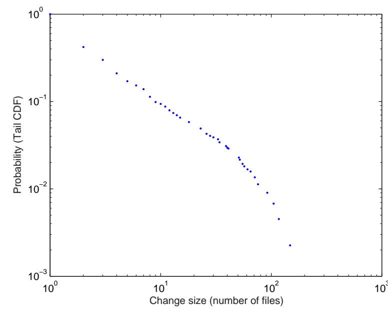
(e) KSDK



(f) KOffice



(g) OpenSSL



(h) Ruby

Figure B.2: Tail CDF of daily system structural changes

Bibliography

[Aca96] Acacia. *The C++ Information Abstraction System*. Website: http://www.research.att.com/sw/tools/Acacia, 1996.

[AL99] Nicolas Anquetil and Timothy Lethbridge. Experiments with clustering as a software remodularization method. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 235–255, Atlanta, Georgia, USA, October 1999.

[Aoy01] Mikio Aoyama. Continuous and discontinuous software evolution: Aspects of software evolution across multiple product lines. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 87–90, Vienna, Austria, September 2001.

[AP01] Annie I. Antón and Colin Potts. Functional paleontology: System evolution as the user sees it. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 421–430, Toronto, Canada, May 2001.

[Axt01] Robert L. Axtell. Zipf distribution of U.S. firm sizes. *Science*, 293:1818–1820, September 2001.

[BE81] L. A. Belay and C. J. Evangelisti. System partitioning and its measure. *The Journal of Systems and Software*, 2:23–29, 1981.

[BE96] Thomas Ball and Stephen G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, April 1996.

[Bel01] Bell Canada. *DATRIX Abstract Semantic Graph Reference Manual, Version 1.4*. Website: http://www.casi.polymtl.ca/casibell, 2001.

[Ber94] Jan Beran. *Statistics for Long-Memory Processes*. Chapman and Hall, 1994.

[BHB99] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a case study: Its extracted software architecture. In *Proceedings of the 21st International Conference on Software Engineering*, pages 555–563, Los Angeles, California, May 1999.

[BKS03] Evelyn J. Barry, Chris F. Kemerer, and Sandra A. Slaughter. On the uniformity of software evolution patterns. In *Proceedings of the 25th International Conference on Software Engineering*, pages 106–113, Portland, Oregon, May 2003.

[BL76] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[BP03] Andreas Bauer and Markus Pizka. The contribution of free software to software evolution. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 170–179, Helsinki, Finland, September 2003.

[BS93] Per Bak and Kim Sneppen. Punctuated equilibrium and criticality in a simple model of evolution. *Physical Review Letters*, 71(24):4083–4086, December 1993.

[BTW87] Per Bak, Chao Tang, and Kurt Wiesenfeld. Self-organized criticality: An explanation of $1/f$ noise. *Pysical Review Letters*, 59(4):381–384, July 1987.

[BTW88] Per Bak, Chao Tang, and Kurt Wiesenfeld. Self-organized criticality. *Pysical Review A*, 38(1):364–374, July 1988.

[BUM02] Binutils User Manual. Website: http://www.gnu.org/software/binutils, 2002.

[Cal88] Jerry Calabaugh. Software configuration – an NP-complete problem. *SIGMIS Database*, 19(2):29–34, 1988.

[CC03] Anthony Cox and Charles Clarke. Syntactic approximation using iterative lexical analysis. In *Proceedings of the 11th International Workshop on Program Comprehension*, pages 154–163, Portland, Oregon, May 2003.

[Cen04] CenterICQ. Website: http://konst.org.ua/en/centericq/, 2004.

[CKN+03] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts, and Kevin Wampler. A system for graph-based visualization of the evolution of software. In *Proceedings of ACM Symposium on Software Visualization*, pages 77–86, San Diego, California, June 2003.

[CNDP04] B. A. Carreras, D. E. Newman, I. Dobson, and A. B. Poole. Evidence for self-organized criticality in a time series of electric power system blackouts. *IEEE Transactions on Circuits and Systems I*, 51(9):1733–1740, September 2004.

[CPP02] CPPX. C++ Source Code Extractor. Website: http://swag.uwaterloo.ca/~cppx, 2002.

[Csc04] Cscope. Website: http://cscope.sourceforge.net, 2004.

[Cta04] Ctags. Website: http://ctags.sourceforge.net, 2004.

[Cub03] Cube. *An Open Source First Person Shooter (FPS) Game Engine*. Website: http://www.cubeengine.com/, 2003.

[Dar59] Charles Darwin. *The Origin of Species by Means of Natural Selection*. 1859.

[Dil04] Dillo. Website: http://www.openoffice.org/, 2004.

[DMM98] D. Doval, Spiros Mancoridis, and Brian Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Proceedings of the International Conference on Software Technology and Engineering Practice*, pages 73–91, Pittsburgh, PA, September 1998.

[EG72] N. Eldredge and S. Gould. Punctuated equilibria: an alternative to phyletic gradualism in models in paleobiology. *Models of Paleobiology*, 1972.

[EGK+01] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J.S. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, January 2001.

[Ema04] Emacs. Website: http://en.wikipedia.org/wiki/Emacs, 2004.

[Epi04] Epiphany. Website: http://www.gnome.org/projects/epiphany/, 2004.

[Ext04] Extreme Programming: A Gentle Introduction. Website: http://www.extremeprogramming.org/, 2004.

[FBMG01] Rudolf Ferenc, Arpad Beszedes, Ferenc Magyar, and Tibor Gyimothy. A short introduction to Columbus/CAN. *Technical Report*, 2001.

[FOP02] K. Fogel, M. O'neill, and M. J. Pearce. CVS log message to change log message conversion script. Website: http://www.red-bean.com/cvs2cl/, 2002.

[FSG04] Rudolf Ferenc, István Siket, and Tibor Gyimóthy. Extracting facts from open source software. In *Proceedings of the 20th International Conference on Software Maintenance*, pages 60–69, Chicago, Illinois, September 2004.

[Gai04] Gaim. Website: http://sourceforge.net/projects/gaim/, 2004.

[GCC02] GCC. GNU Compiler Collection. Website: http://gcc.gnu.org, 2002.

[Ger91] Connie J. Gersick. Revolutionary change theories: A mutilevel exploration of the punctuated equilibrium paradigm. *Academy of Management*, 16(1):10–36, January 1991.

[GG05] Alan Grosskurth and Michael W. Godfrey. A reference architecture for web browsers. In *Proceedings of the 21th International Conference on Software Maintenance*, Budpest, Hungary, September 2005.

[GGP03] Edoardo Gaffeo, Mauro Gallegati, and Antonio Palestrini. On the size distribution of firms: Additional evidence from G7 countries. *Physica A*, 324:117–123, 2003.

[GJKT97] Harald Gall, Mehdi Jazayeri, René Klöesch, and Georg Trausmuth. Software evolution observations based on product release history. In *Proceedings of the 13th International Conference on Software Maintenance*, pages 160–166, Bari, Italy, October 1997.

[GJR99] Harald Gall, Mehdi Jazayeri, and Claudio Riva. Visualizing software release histories: The use of color and third dimension. In *Proceedings of the 15th International Conference on Software Maintenance*, pages 99–108, Oxford, England, September 1999.

[Gnu03] Gnumeric. Website: http://www.gnome.org/projects/gnumeric/, 2003.

[Goo00] Richard Gooch. *Linux Kernel API Changes from 2.0 to 2.2*. Website: http://www.atnf.csiro.au/people/rgooch/linux/docs/porting-to-2.2.html, 2000.

[GP95] David Garlan and Dewayne Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4):269–274, April 1995.

[GT00] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Proceedings of the 16th International Conference on Software Maintenance*, pages 131–142, San Jose, California, October 2000.

[GXL02] GXL. *The Graph eXchange Language*. Website: http://www.gupro.de/GXL, 2002.

[GY03] A. A. Gorshenev and Yu. M. Pis'mak. Punctuated equilibrium in software evolution. *DOI: cond-mat/0307201*, July 2003.

[HHL+00] Ahmed E. Hassan, Richard C. Holt, Bruno Lague, Sebastien Lapierre, and Charles Leduc. E/R schema for the DATRIX C/C++/Java exchange format. In *Proceedings of the 7th Working Conference on Reverse Engineering*, pages 284–286, Brisbane, Australia, November 2000.

[Hig99] J. A. Highsmith. *Adaptive Software Development – A Collaborative Approach to Managing Complex Systems*. Dorset House, New York, NY, USA, 1999.

[HJ02] Francis Hunt and Paul Johnson. On the Pareto distribution of Sourceforge projects. In *Proceedings of Open Source Software Dvelopment Workshop*, pages 122–129, Newcastle, UK, February 2002.

[Hol02] Richard C. Holt. *An Introduction to TA: the Tuple-Attribute Language*. Website: http://plg.uwaterloo.ca/~holt/cv/papers.html, 2002.

[Hur51] H. E. Hurst. Long-term storage capacity of reservoirs. *Transactions of American Society of Civil Engineers*, 116:770–799, 1951.

[HWH05] Ahmed E. Hassan, Jingwei Wu, and Richard C. Holt. Visualizing historical data using spectrographs. In *Proceedings of the 11th International Software Metrics Symposium*, Como, Italy, September 2005.

[id 03] id Software. *Quake – A First Person Shooter Computer Game*. Website: http://www.idsoftware.com/, 2003.

[IEE00] IEEE. Recommended practice for architectural description of software-intensive systems. *IEEE Std 1471*, 2000.

[Inn04] InnoDB. Website: http://www.innodb.com, 2004.

[Jen98] H. J. Jensen. *Self-Organized Criticality - Emergent Complex Behavior in Physical and Biological Systems*. Cambridge University Press, 1998.

[JGr04] JGrok. A Query Language for Reverse Engineering. Website: http://swag.uwaterloo.ca/tools.html, 2004.

[JS03] C. Jensen and W. Scacchi. Simulating an automated approach to discovery and modeling of open source software development. In *Proceedings of Software Process Simulation and Modeling Workshop*, Portland, OR, USA, May 2003.

[KC98] Rick Kazman and S. Jeromy Carrière. View extraction and view fusion in architectural understanding. In *Proceedings of the 5th International Conference on Software Reuse*, pages 290–299, Victoria, BC, Canada, June 1998.

[KDE04] KDE. The K Desktop Environment. Website: http://www.kde.org, 2004.

[KE00] Rainer Koschke and Thomas Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 201–210, Limerick, Ireland, June 2000.

[Koc04] Stefan Koch. Profiling an open source project ecology and its programmers. *Electronic Markets*, 14(2):77–88, July 2004.

[KOf04] KOffice. Website: http://www.koffice.org, 2004.

[Kon04] Konqueror. *A File Manager and Web Browser for KDE.* Website: http://www.konqueror.org/, 2004.

[Kop04] Kopete. Website: http://kopete.kde.org/, 2004.

[KS96] Chris F. Kemerer and Sandra Slaughter. Need for more longitudinal studies of software maintenance. In *Proceedings of the International Workshop on Empirical Studies of Software Maintenance*, Monterey, California, USA, 1996.

[Lan01] Michele Lanza. The evolution matix: Recovering software evolution using software visualization techniques. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 37–42, Vienna, Austria, September 2001.

[Law82] M. J. Lawrence. An examination of evolution dynamics. In *Proceedings of the 6th International Conference on Software Engineering*, pages 188–196, Tokyo, Japan, September 1982.

[LB85] M. M. Lehman and L. A. Belady. *Program Evolution – Processes of Software Change*. Academic Press, London UK, 1985.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, September 2003.

[Leh97] M. M. Lehman. Laws of software evolution revisited. *Lecture Notes in Computer Science*, 1149:108–124, 1997.

[LG97] Arun Lakhotia and John M. Gravley. A unified framework for expressing software subsystem classification techniques. *The Journal of Systems and Software*, 36(3):211–231, March 1997.

[LHM03] Yuan Lin, Richard C. Holt, and Andrew Malton. Completeness of a fact extractor. In *Proceedings of the 10th Working Conference on Reverse Engineering*, pages 196–205, Victoria, British Columbia, Canada, November 2003.

[Lin04] Linux Kernel. Website: http://www.kernel.org, 2004.

[LR01] M. M. Lehman and J. F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11(1):15–44, November 2001.

[LRW+97] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. In *Proceedings of the 4th International Software Metrics Symposium*, pages 20–32, Albuquerque, NM, November 1997.

[LS91] Ronald Lange and Robert W. Schwanke. Software architecture analysis: A case study. In *Proceedings of the 3rd international workshop on Software configuration management*, pages 19–28, Trondheim, Norway, June 1991.

[Lyn04] Lynx. *A Text Browser for World Wide Web*. Website: http://www.openoffice.org/, 2004.

[Man82] Benoit B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman & Company, 1982.

[Man01] Manifesto for Agile Software Development. Website: http://agilemanifesto.org/, 2001.

[MFH02] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, July 2002.

[MFT02] Greg Madey, Vincent Freeh, and Renee Tynan. The open source software development phenomenon: An analysis based on social network theory. In *Proceedings of Americas Conference on Information Systems*, pages 1806–1813, Dallas, TX, USA, 2002.

[Mil02] E. Milotti. 1/f Noise: a Pedagogical Review. *ArXiv Physics e-prints*, April 2002.

[Mir04] Miranda. Website: http://www.miranda-im.org/, 2004.

[MM01] Brian Mitchell and Spiros Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *Proceedings of the 17th International Conference on Software Maintenance*, pages 744–753, Florence, Italy, November 2001.

[MMCG99] Spiros Mancoridis, Brian Mitchell, Yihfarn Chen, and Emden Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the 15th International Conference on Software Maintenance*, pages 50–59, Oxford, England, September 1999.

[MN96] Gail C. Murphy and David Notkin. Lightweight lexical source model exraction. *IEEE Transactions on Software Engineering*, 5(3):262–292, July 1996.

[MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28, New York, NY, USA, October 1995.

[Moo01] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 13–22, Suttgart, Germany, October 2001.

[MOTU93] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse engineering approach to system structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, December 1993.

[Moz04] Mozilla Application Suite. Website: http://www.mozilla.org, 2004.

[MyS04] MySQL. Website: http://www.mysql.com, 2004.

[New05] M. E. J. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46(5):323–351, September 2005.

[Nor03] Martin E. Nordberg. Managing code ownership. *IEEE Software*, 20(2):26–33, January 2003.

[NYN+02] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwe Ye. Evolution patterns of open-source software systems and communities. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 76–85, Orlando, Florida, May 2002.

[O'M06] S. O'Mahony. Managing community software in a commodity world. In *Chapter 8 of Frontiers of Capital: Ethnographic Reflections on the New Economy.* Duke University Press, forthcoming 2006.

[Ope04a] OpenSSH. Website: http://www.openssh.org, 2004.

[Ope04b] OpenSSL. Website: http://www.openssl.org, 2004.

[Ope05a] Open Source Definition. Website: http://www.opensource.org/docs/definition.php, 2005.

[Ope05b] OpenOffice. Website: http://www.openoffice.org/, 2005.

[Par94] David Lodge Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, pages 279–287, Sorrento, Italy, May 1994.

[Per02] Dewayne E. Perry. Law and principles of evolution. In *Proceedings of the 18th International Conference on Software Maintenance*, page 70, Montreal, Canada, October 2002.

[PHP04] PHP. Hypertext preprocessor. Website: http://www.php.net, 2004.

[Pos03] PostgreSQL. Website: http://www.postgresql.org, 2003.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[RAGBH05] Gregorio Robles, Juan José Amor, Jesús M. Gonzléz-Barahona, and Israel Herraiz. Evolution and growth in large libre software projects. In *Proceedings of the International Workshop on Principles of Software Evolution*, Lisbon, Portugal, September 2005.

[Ray99] Eric S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly and Associates, 1999.

[Rub04] Ruby. Website: http://www.ruby-lang.org, 2004.

[RY81] Vijay V. Raghavan and C. Yu. A comparison of the stability characteristics of some graph theoretic clustering methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3(4):393–402, August 1981.

[Saf04] Safari. Website: http://www.apple.com/macosx/features/safari/, 2004.

[San95] Georg Sander. VCG – visualization of compiler graphs. *Technical Report A01-95*, February 1995.

[San03] Anand K. Santhanam. *Towards Linux 2.6: A Look into the Next Kernel*. Website: http://www.ibm.com/developerworks/linux/library, 2003.

[SBM+02] Margaret-Anne Storey, Casey Best, Jeff Michaud, Derek Rayside, Marin Litoiu, and Mark Musen. Shrimp views: An interactive environment for information visualization and navigation. In *Proceedings of Conference on Human Factors in Computing Systems*, pages 520–521, Minneapolis, Minnesota, USA, April 2002.

[Sch90] C. H. Scholz. *The Mechanics of Earthquakes and Faulting*. Cambridge University Press, 1990.

[Sch91] Robert W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th International Conference on Software Engineering*, pages 83–92, Austin, Texas, United States, May 1991.

[SHE02] Susan Elliott Sim, Richard C. Holt, and Steve Easterbrook. On using a benchmark to evaluate c++ extractors. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 114–123, Paris, France, June 2002.

[SMBB97] R. V. Solé, S. C. Manrubia, M. Benton, and P. Bak. Self-similarity of extinction statistics in the fossil record. *Nature*, 388(21):764–767, August 1997.

[SO02] Stephen R. Schach and A. Jefferson Offutt. On the nonmaintainability of open source software. In *Proceedings of the 2nd Workshop on Open Source Software Engineering*, Orlando, Florida, May 2002.

[Sou05] SourceForge. Website: http://sourceforge.net/, 2005.

[Spr03] J. C. Sprott. *Chaos and Time-Series Analysis*. Oxford University Press, 2003.

[Swa02] SwagKit. *The Software Architecture Group (SWAG) Analysis Toolkit*. Website: http://swag.uwaterloo.ca/swagkit, 2002.

[TGLH00] John B. Tran, Michael W. Godfrey, Eric H. S. Lee, and Richard C. Holt. Architecture repair of open source software. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 48–59, Limerick, Ireland, June 2000.

[TH99] Vassilios Tzerpos and Richard C. Holt. MoJo: A distance metric for software clusterings. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 187–193, Atlanta, Georgia, USA, October 1999.

[TH00a] Vassilios Tzerpos and Richard C. Holt. ACDC: An algorithm for comprehension driven clustering. In *Proceedings of the 7th Working Conference on Reverse Engineering*, pages 258–267, Brisbane, Australia, November 2000.

[TH00b] Vassilios Tzerpos and Richard C. Holt. On the stability of software clustering algorithms. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 211–218, Limerick, Ireland, June 2000.

[TkS03] TkSee/SN. *A C++ Source Code Extractor Based on Cygnus Source Navigator*. Website: http://www.site.uottawa.ca/~tcl/kbre, 2003.

[TM02] Christopher M. B. Taylor and Malcolm Munro. Revision towers. In *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, pages 43–50, Paris, France, June 2002.

[Tur02] Wladyslaw M. Turski. The reference model for smooth growth of software systems revisited. *IEEE Transactions on Software Engineering*, 228(8):814–815, August 2002.

[WH04a] Jingwei Wu and Richard C. Holt. Linker-based program extraction and its uses in studying software evolution. In *Proceedings of the International Workshop on Foundations of Unanticipated Software Evolution*, pages 1–15, Barcelona, Spain, March 2004.

[WH04b] Jingwei Wu and Richard C. Holt. Resolving linkage anomalies in extracted system models. In *Proceedings of the 12th International Workshop on Program Comprehension*, pages 241–245, Bari, Italy, June 2004.

[WHH04] Jingwei Wu, Richard C. Holt, and Ahmed E. Hassan. Exploring software evolution using spectrographs. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 80–89, Delft, Netherlands, November 2004.

[Wig97] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the 4th Working Conference on Reverse Engineering*, pages 33–43, Amsterdam, The Netherlands, October 1997.

[WSHH04] Jingwei Wu, Claus W. Spitzer, Ahmed E. Hassan, and Richard C. Holt. Evolution spectrographs: Visualizing punctuated change in software evolution. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 57–66, Kyoto, Japan, September 2004.

[WT04] Zhihua Wen and Vassilios Tzerpos. Evaluating similarity measures for software decompositions. In *Proceedings of the 20th International Conference on Software Maintenance*, pages 368–377, Chicago, IL, USA, September 2004.

[ZG03] Lijie Zou and Michael W. Godfrey. Detecting merging and splitting using origin analysis. In *Proceedings of the 10th Working Conference on Reverse Engineering*, pages 146–154, Victoria, BC, Canada, November 2003.

[ZW04] Thomas Zimmermann and Peter Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR)*, pages 2–6, Edinburgh, UK, May 2004.