Seeking Empirical Evidence for Self-Organized Criticality in Open Source Software Evolution

Jingwei Wu [1] and Richard Holt [2]

Software Architecture Group (SWAG)
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada

Abstract

We examine eleven open source software systems and present empirical evidence for the existence of fractal structures in software evolution. In our study, fractal structures are measured as power laws through the lifetime of a software system. We describe two specific power law related phenomena: the probability distribution of software changes decreases as a power function of change sizes; and the time series of software change exhibits long range correlations with power law behavior. The existence of such spatial (across the system) and temporal (over the system lifetime) power laws suggests that *Self-Organized Criticality* (SOC) occurs in the evolution of open source software systems. As a consequence, SOC may be useful as a conceptual framework for understanding software evolution dynamics (the cause and mechanism of change or growth). We give a qualitative explanation of software evolution based on SOC. We also discuss some potential implications of SOC to current software practices.

1 Introduction

Laws of software evolution formulated by Lehman [Leh97] represent a major intellectual contribution to understanding software evolution dynamics (the underlying cause and mechanism of software 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]. The laws suggest that software systems must be continuously adapted to respond to external forces such as new functional requirements and hardware upgrade and to maintain stakeholder satisfaction.

[1] Email: j25wu@uwaterloo.ca
[2] Email: holt@uwaterloo.ca

Viewing software evolution as a phenomenon driven by various external forces can make it difficult to accommodate and explain conflicting findings across different software systems or domains. For example, open source projects in different domains have been found to grow at different rates including super-linear, sub-linear and linear rates [GT00][RAGBH05]. A simple, unified way for explaining evolution dynamics is needed. This paper looks into *Complex Systems* theories to seek new ways for describing and explaining 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 event size can control their changes or responses over time. That is, changes in a complex system can be of any size and occur any time. For example, a power blackout can strike a street, a town, a state and all the way up to a country. An intriguing aspect of complex 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 typical power law behavior [BTW87]. SOC was set out as an ambitious effort for explaining the existence of ubiquitous fractal structures in nature. SOC has two important signatures: (1) the power law distribution of dynamical responses, and (2) long range correlations with power law behavior in time series of response [CNDP04][SMBB97]. Section 2 will provide a brief introduction to fractals, power laws and long range correlations and explain how these concepts are relevant to SOC.

Does a software system follow the SOC dynamics during its evolution? This question arises from several previous studies which in particular include:

- Software evolution as punctuated equilibrium
  Wu *et. al* examined the structural evolution of software systems at the level of source files. They observed that three open source systems (OpenSSH, PostgreSQL and Linux Kernel) evolved through an alternation between long periods of small changes and short periods of large avalanche changes [WSHH04].
- Biological evolution as self-organized criticality
  Researchers have found that fluctuations in fossil record exhibit 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 such as mean and standard variance remain the same as those obtained from the whole time series if a change of scale is performed. SOC is 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.

- Overall open source movement as a self-organizing collaborative social network
  In comparison to traditional industrial systems, open source software systems are largely developed based on a less strict control and management model [Ope05][Ray99]. Spontaneous collaboration is promoted and backed by a decentralized developer community across the Internet [Ray99]. Researchers including Madey and Koch have suggested that open source projects can be seen as a self-organizing phenomenon featuring the self-selection of tasks, spontaneous collaboration and leadership [Koc04][MFT02]. The main empirical evidence they present is 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).

Motivated by these promising results, we feel that SOC may be established as a useful conceptual framework for describing and explaining software evolution. In this paper, we describe our effort in seeking evidence for SOC in open source software systems and also discuss some implications of SOC to software practices.

The rest of this paper is organized as follows. Section 2 introduces useful terms and concepts in brief. Section 3 explains empirical data collection in regard to software changes and time series of change. Section 4 investigates the existence of power laws in the evolution of open source software systems. Section 5 provides a qualitative explanation of software evolution based on SOC. Section 6 discusses some threats to the validity of our work. Section 7 considers related work and Section 8 concludes this paper.

2 Background

This section provides a brief introduction to fractals, power laws, R/S time series analysis and SOC. The reader familiar with these concepts can skip to the next section.

2.1 Fractals

Fractals are mathematical or natural objects that are made of parts similar to the whole in "some" way. A fractal has a self-similar structure that occurs at different scales. For example, 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. For a detailed explanation of fractals, the reader can refer to Mandelbrot's book *Fractal Geometry of Nature* [Man82].

2.2 Power Law

A power law is a relationship between two scalar variables x and y, which 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 equivalent 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 equation $f(x) = C \cdot x^k$ has a property that relative scale 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 because of its scale invariance.

Power Law Distribution

This paper 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 size s is specified as $D(s) = P(x \geq s) \sim s^{-\beta}$. The relationship between α and β is $\beta = \alpha - 1$ [New05]. Because β can be conveniently estimated using linear regression on a log-log plot without binning data points, we choose to estimate β rather than α in our study.

Power laws have been observed in many fields such as physics, economics, geography and sociology [New05]. For example, the distribution of earthquakes is found to follow $P(E) \sim E^{-\alpha}$ 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 (measured as the number of employees) [Axt01][GGP03] and of open source project sizes (measured as the number of developers or lines of code) [HJ02][Koc04] also follow power laws.

2.3 Time Series Analysis

Time series are 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, which 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 paper, 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, also referred to as the R/S statistic. It was formulated by Hurst in 1951 [Hur51]. Hurst 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 defined the R/S statistic which calculates the power law relationship between rescaled adjusted range $R/S(\tau)$ and time lag τ:

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

where H is often 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. It is defined 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 standard deviation obtained for the ith block of τ data points. If the τ data points from the ith block are re-numbered as $\{x_1, x_2, ..., x_\tau\}$, $R_i(\tau)$ and $S_i(\tau)$ are calculated as follows:

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

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

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

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

The Hurst exponent H can reflect data persistence in a time series. Based on the value of H, natural and 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 $H > 0.5$. A processes from this category has long runs of consecutive values above or below the mean. The Nile River has $H = 0.91$ as calculated by Hurst [Hur51]. This value implies that flood occurrences of the Nile River are not purely random but temporally dependent. Such phenomena are often referred to as long range dependence or long range correlations.
- *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 [Ber94].

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 a quantification allows us to study similarities between different temporal processes (including software evolution), thereby recognizing underlying unifications that might otherwise have gone unnoticed.

Long range correlations in a time series can also be analyzed using the 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 [3] [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.

2.4 Self-Organized Criticality

In 1987, Bak, Tang and Wiesenfeld proposed Self-Organized Criticality (SOC) to explain widespread occurrences of spatial fractals and fractal time series (also known as $1/f$ noise) in nature [BTW87]. According to SOC, a complex system which consists of interacting components can exhibit some general characteristic behavior in time ($1/f$ noise) and in space (self-similar fractals) spontaneously. Such behavior is measured as power laws.

The $1/f$ noise is a ubiquitous phenomenon [Mil02]. For a given system, we obtain a time series (signal) by measuring one of its time-dependent properties. 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 two most important diagnostics of SOC and are commonly measured as power laws

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

[BTW88]. If a system exhibits power laws without apparent tuning then it is said to follow the SOC dynamics.

3 Data Collection

We now describe how to collect software changes throughout the lifetime of a software system.

3.1 Software Change

We denote a software change as a set of source files that are modified together for a purpose. For example, a change may contain a number of source files which a developer modified in order to remove a defect or add a feature. We consider two kinds of change: *Logical Changes* recovered from source control repositories (*e.g.*, CVS) and *Structural Changes* obtained by contrasting subsequent snapshot versions (*e.g.*, releases). For the C and C++ programming language, we consider files with extensions as follows: `.c`, `.C`, `.cc`, `.cpp`, `cxx`, `.c++`, `.h`, `.H`, `.hh`, `.hpp`, `.hxx` and `.h++`. The size of a change is measured as the number of source files that are contained in the change.

Logical Change
A logical change, which is recoverable from a source control repository, contains files checked in by the same developer with the same log message and at the same time. The term "same time" in this context means that files are committed in a short period. Zimmermann *et al.* described a number of methods for recovering logical changes from the CVS repository [ZW04]. One of their methods is 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 choose to use *cvs2cl* in our study. Recovered changes may be related to different types of task such as bug fix, feature modification, functional improvement and refactoring. For simplicity they are not differentiated for our work.

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. As shown in Figure 1, a delta graph contains five files B, C, D, F, G and H as well as six added and deleted dependencies among these files. 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. File D forms a structural change by itself.

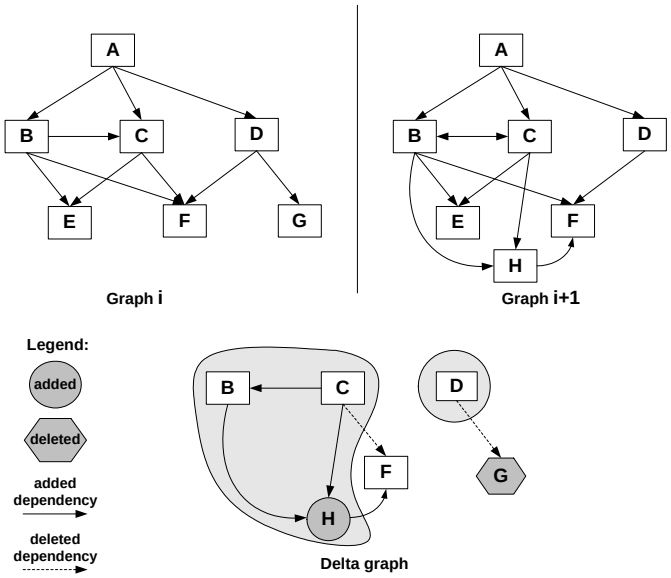


Fig. 1. Recover Structural Changes

We use CTSX [WH06], a robust and efficient program extractor for C and C++, to extract structural dependency graphs on a daily basis over the lifetime of a software system. We then compare consecutive graphs to recover structural changes.

3.2 Time Series of Change

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

4 Fractals in Software Evolution

This section presents empirical evidence for fractal structures found in the evolutionary history of eleven open source software systems. The following fractal related phenomena are our main concern:

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

We first discuss in detail the empirical results obtained from GCC (GNU Compiler Collection) [GCC02]. We then summarize empirical results from more software systems, which include NetBSD, FreeBSD, OpenBSD, Linux,

PostgreSQL, KSDK, KOffice, PHP, OpenSSL and Ruby. Appendix A provides a brief introduction to these systems.

4.1 Power Law Distribution of Change Sizes

In our analysis of distributions of change sizes for GCC, the quantity being plotted is $D(s)$, which is the tail cumulative distribution function (see Section 2.2).

4.1.1 Distribution of Logical Changes in GCC

We recovered a total of 40,034 logical changes from the CVS repository of GCC. The studied development period is about eight years, from 1997/08/11 to 2005/09/09. Fig. 2(a) displays $D(s)$ on a log-log plot, which follows approximately a straight line. An ordinary least squares (OLS) linear estimation on the logarithmic scale of base 10 yields the following:

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

$D(s)$ has a property that relative change $D(ks)/D(s) = k^{-\beta}$ is independent of size s. $D(s)$ is thereby scale invariant and it can be seen as a self-similar object statistically. In GCC, a larger logical change occurs less frequently than a smaller one. We can estimate how much rarer large changes are using the obtained $D(s)$ equation. In Fig. 2(a) the probability of a change occurrence that involves more than 100 source files is extremely low, roughly less than 0.16%.

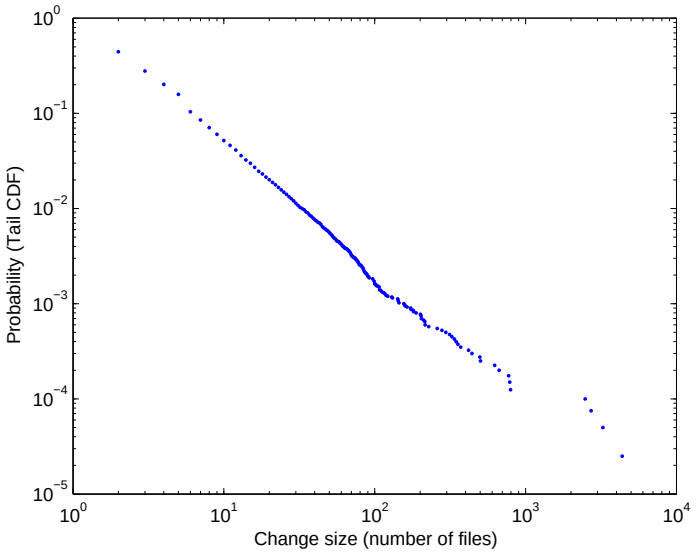
We have also found that similar power law distributions hold for individual years from 1998 to 2005 in GCC. The obtained scaling exponents vary from 1.29 to 1.34. This suggests that yearly distributions share a similar power law behavior with each other and also with the lifetime distribution.

4.1.2 Distribution of Structural Changes in GCC

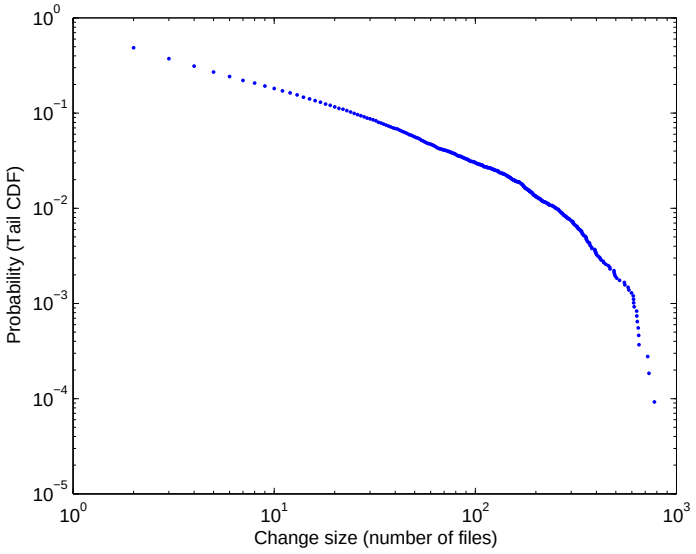
After observing that the size distribution of logical changes followed a power law relationship, we were curious to know whether other kinds of software change have a similar distribution. We examined the size distribution of daily structural changes over the same development period of GCC. A power law was observed by neglecting changes consisting of more than 100 files. It is shown as a log-log plot in Fig. 2(b). An OLS linear fit on the logarithmic scale gives the following:

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

The exponent β is estimated with $1 \leq s \leq 100$. 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. Such a large change may



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



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

Fig. 2. Tail Cumulative Distributions of Change Sizes for GCC

structurally depend on hundreds of other files in the system. It is difficult to modify a substantial number of dependencies among several hundred files.

The long tail deviation from a power law may be relevant to the finite-size effect [BTW88]. We suspect that the deviation might appear at a larger change size if the number of files in GCC grows significantly in the future. We have

System	Period	Logical change (LC)			Structural change (SC)		
		#LC	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

$*$: Structural changes of Linux were obtained by comparing consecutive releases over time.

Table 1
Scaling Exponents for Distributions of Software Changes

also 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 equivalent size. For Linux which is many times larger than GCC, the deviation appears around $s = 350$. These differences can be seen in log-log plots shown in Appendix B.

4.1.3 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 1. The obtained log-log plots are shown in Appendix B.

We did not analyze distributions of structural change sizes for three BSD variants because each of them is actually composed of a large number of smaller applications and libraries. It is beyond the scope of this paper to study the evolution of structural dependencies among a collection of interacting applications. Unlike structural changes, logical changes are mostly within the boundary of each smaller application or library. Thus we only studied distributions of logical changes for three BSDs.

Because Linux does not have a source control repository such as CVS for public access, we were not able to analyze change log information or perform daily structural comparisons. We instead examined the structural evolution of Linux through comparing 524 releases (from 1.0 to 2.6.12.3). These releases

were chosen and ordered according to release dates to create a historical sequence, which contains both stable and experimental releases.

Here are some of our observations on the studied open source systems.

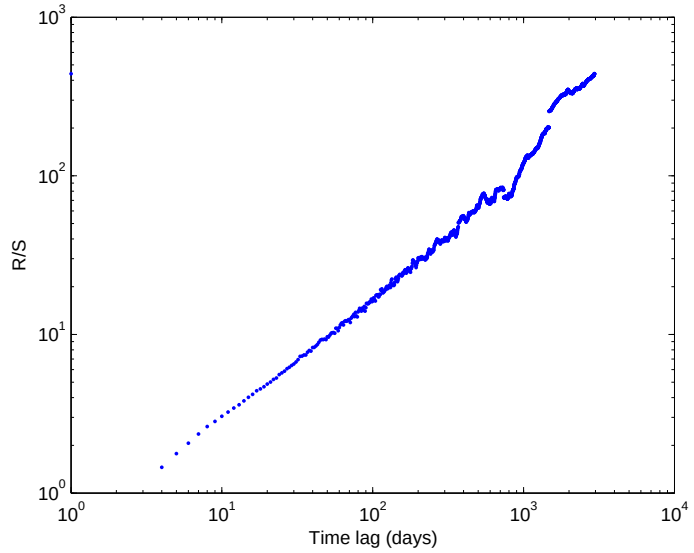
- All eleven software systems we studied have a power law distribution with respect to either logical or structural changes. The quality of fit (R^2) values shown in Table 1 indicate a strong linear relationship between function $D(s)$ and the number of source files on the logarithmic scale.
- The distribution of logical changes has a larger scaling exponent than distribution of structural changes. This is limited to a size threshold, above which a large structural change can have a lower probability if compared to a logical change of the same size. This can be seen from Figure 2. The distribution in Fig. 2(b) deviates from the power law at larger sizes ($s > 100$) and drops toward probability zero. 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 logical changes.
- OpenBSD has a scaling exponent different from those of FreeBSD and NetBSD. This suggests that products from the same product family may exhibit slightly different behaviors. This perhaps is because that FreeBSD and NetBSD have a longer history and thus have more logical changes than OpenBSD. Further examination will be needed for understanding what causes such differences in β from system to system.
- For Linux, time intervals between any two adjacent releases vary 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 in fact favors large structural changes rather than small ones. As a result, Linux has the smallest scaling exponent.

4.2 Long Range Correlations in Time Series

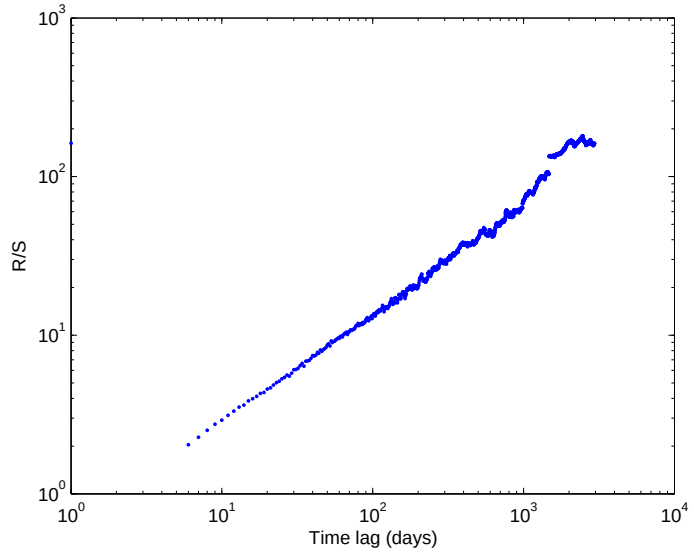
The essential nature of software system evolution is change occurring spatially (across the system) and temporally (over the system lifetime). Our observation of scale invariance in distributions of change sizes leads us to wonder wether the evolution of a software system exhibits fractal structures in time, *i.e.*, long range correlations with power law behavior. As pointed out in Section 2.3, the R/S analysis can be used to analyze long range correlations in time series. Therefore, to answer the above question, we need to determine whether a time series of software change has a Hurst exponent greater than 0.5 (the characteristic value of random noise).

4.2.1 R/S Analysis of GCC

The two R/S statistic plots shown in Figure 3 have Hurst exponents with $H = 0.7711$ for time series of logical change and $H = 0.6841$ for time series of



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



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

Fig. 3. R/S Analysis of Daily Time Series for GCC

structural change. These exponents are significantly above 0.5, thus indicating strong long range correlations. These 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 a reduction of H towards 0.5.

System	Logical 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 was obtained by comparing consecutive releases over time.

Table 2
Hurst Exponents from R/S Analysis of Daily Time Series

4.2.2 R/S Analysis of More Systems

We performed the R/S analysis on the same set of systems we examined in Section 4.1. We estimated Hurst exponents by considering time lags smaller than 365 (days). The largest time lag for Linux is 174 (releases). This is because Linux's time series were obtained by comparing subsequent releases but not daily snapshot versions. The time lag 174 accounts approximately for one third of 524 Linux releases we studied. The obtained Hurst exponents are summarized in Table 2.

We also defined *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. Table 2 shows that Ruby has the smallest TSC (14.7%) with regard to structural change activities, indicating 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 logical and structural changes. Larger systems such as GCC, Linux, KOffice and three BSD variants tend to change every day. This is not surprising because more developers are usually involved in a larger project and change activities occur more frequently.

The H values obtained for all the time series of logical change vary between 0.7 and 0.8. This indicates that these time series appear approximately equally correlated in the long run. By contrast, all the H values obtained for structural change have a wider span from 0.6 to 0.8 with Ruby excluded. The Hurst exponent for 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 notable correlations.

4.3 Summary

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

5 Discussion

All the eleven systems we examined have evolved over many years. However, their change dynamics share a typical power law behavior, which appears to be independent of individual details of each system. Where do these scale free dynamics come from? We provide a qualitative explanation of software evolution dynamics from a perspective based on SOC.

Bak used the sandpile model to illustrate the dynamics of a SOC system [BTW88]. Suppose a person starts to build a sandpile on a flat square board by means of dropping grains of sand randomly, one grain at a time. The whole sandpile grows as many smaller piles are formed and their slopes increase continuously. The slopes at different locations eventually reach a critical value. If more sand is added, sand slides 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; 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 in the evolution of open source systems (see Section 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 a software system in a qualitative way. Table 3 shows our proposed analogy, which has four elements: *driving force*, *response*, *system state*, and *relaxing force*. We now interpret their meanings.

Like a sandpile, an evolving software system is continuously changed under the influence of various driving forces as diverse as new customer requirements, hardware upgrade, and development process. More specifically, changes occur in response to requests related to bugs, refactorings, features and etc. Changes

propagate to different locations within the system. From our point of view, a change request is analogous to a sand drop and a (logical or structural) change is mapped to a sand slide.

In the sandpile model, system state is a matrix of maximum gradients (slopes) covering every location occupied by sand, and relaxing force is gravity which controls sand slides by reducing maximum gradients at appropriate locations if more grains of sand are dropped. We map stakeholder satisfaction to gravity and the release/iteration plan to the sand gradient profile. The demands from different stakeholders such as developers, architects and customers must be satisfied. This is consistent with Lehman's law of *Continuing Change* [Leh97] stating that changes must be made continuously to maintain stakeholder satisfaction. The relaxing force can be 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. Changes are estimated, planned and performed at different levels of priority and severity. For example, a refactoring is often needed after many new features are added. In order to perform an assigned or self-selected refactoring, a developer (stakeholder) builds a progressive understanding of unfamiliar parts through communication with other developers and performs the refactoring task incrementally. After the task is done, the developer re-gains satisfaction with the system.

Criticality and Self-Organization

When does an evolving software system enter criticality and what does self-organization mean in the context of software evolution? We now offer our answers to these two questions.

We studied eleven open source systems in total 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 distributions of change sizes and in time series of change. We thus suspect that an evolving software system may enter criticality no later than its first release.

To understand self-organization, we need to take into consideration de-

	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 satisfaction

Table 3
Sandpile Model and Software System

veloper behaviors and development processes. In large open source projects, developers commonly collaborate with one another for some common purposes spontaneously and they are free to modify and redistribute the source code and to work on the segments of their own interest. A central organization in the name of core group or steering committee [BP03] may exist. It plays an important role in providing general development guidelines or road maps, anchoring a broader community of developers and users, and nurturing leadership and collaboration across the community [Hig99]. However, such an organization does not control or command what and how an individual developer should modify the source code. Spontaneous collaboration activities between developers ensure the delivery of a system's first release and then sustain the future evolution of the system.

Self-organization shall not be confused with Lehman's third law of software evolution, *Self-Regulation* [Leh97]. Self-regulation is a control notion suggesting that both positive and negative feedback controls are constantly and pervasively exercised during a system's 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 software process guideline [JS03].

Change Propagation

If a software system follows the SOC dynamics, can we forsee the extent to which changes propagate through the system? According to SOC, a complex system is evolving at the edge between chaos and order where no single characteristic size can control the evolution of the system [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 small town [CNDP04]. It is difficult to predict where a blackout can occur and how far the blackout can propagate.

As a software system evolves to respond 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 have observed in the evolution of open source systems, a change can be up to any sizes. The occurrences of change covering a significant part of a system or even the entire system, though rare, appear inevitable in open source projects. Necessary measures should be adopted to facilitate communication and collaboration between developers to prepare them for a large unexpected change. For instance, *Collective Code Ownership* [Nor03] offers an effective strategy for facilitating collaboration among developers and improving the quality of code.

Agile Software Development

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

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

These 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 models in today's turbulent e-business world [Hig99]. His idea is drawn from theories for complex adaptive systems (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 in real life. Other agile methodologies share similar ideas.

Agile software development has grown in recognition of complex systems theories which value adaptation over prediction or optimization [Man01]. The agile manifesto was summarized by many agile software practitioners based on their hands-on experience in developing and managing large numbers of real life 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 Validity Threats and Limitations

There are several threats to the validity of our work.

- The eleven systems we studied are all successful, large open source software systems. There are large numbers of small open source projects which neither attract many developers nor are maintained actively over a long period of time. Therefore, the systems we studied are not representative of small or failed open source projects.
- It is unknown whether closed source industrial systems exhibit similar power laws in space and in time. This needs to be empirically checked through future work.
- Structural change is not as clear cut as logical change. The definition and identification of a structural change is affected by how one interprets changes to the system structure and how frequently structural snapshots are captured. Furthermore, CTSX that we used to extract program structural dependencies produces significant inaccuracies when handling C++ programs. This perhaps explains why KOffice implemented in C++ does not to follow power laws during its structural evolution (see Fig. B.1(f)).
- Some simulation models for SOC [BS93][SMBB97] can be adapted to sim-

ulate software evolution and verify our empirical findings from a theoretical point of view. For example, Cook *et. al* have presented a conditional growth model to reveal the existence of SOC in software growth [CHW05]. By adapting their model to software changes, we might be able to obtain theoretical proof of SOC in software evolution.

7 Related Work

In several empirical studies of software evolution, researchers have examined power law distributions related to open source projects hosted on the SourceForge Web site [Sou05]. The mission of SourceForge is to enrich the open source community by providing a centralized place for open source developers to 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 a rank-based power law distribution [New05]. According to their findings, there are a small number of exceptionally popular open source projects such as Linux, Apache and Mozilla while most SourceForge projects are less popular. They suggest that studying median size projects instead of exceptionally popular projects might be useful for identifying best practice for developing open source software.

Madey *et al.* reported similar power law distributions about open source projects [MFT02]. They choose to measure project sizes using 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 movement 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 different development teams. Madey found that cluster sizes also followed power laws. The presence of power laws is suggestive evidence that open source software development can be modeled as self-organizing collaborative social networks.

Koch studied individual programmers' contribution to open source projects at SourceForge and he found power laws [Koc04]. Koch measured the contribution of a programmer using the number of commits made by the programmer or the number of projects the programmer worked on. The observed power laws indicate that a minority of programmers are in fact responsible for the major growth of open source projects. In addition, the collocation of projects in a virtual hosting environment such as SourceForge does not significantly increase co-participation over different projects.

These studies have treated the open source software community as an ecology in which individual programmers collaborate with one another, ideas are nurtured, and projects are delivered. Both Madey and Koch have suggested

that the overall open source software development is a self-organizing system in which spontaneous collaboration as well as leadership (*e.g.*, chief programmers) contribute to the success of many open source projects. By contrast, we examined power laws during the evolution of several open source systems and suggested that software systems follow the SOC dynamics.

The work of Gorshenev and Pis'mak on explaining software evolution dynamics based on SOC perhaps is the most relevant to our work [GY03]. They observed that distributions of added lines (or deleted lines) follow power laws in three open source systems which are Mozilla, FreeBSD and Emacs. We studied logical and structural changes at the source file level and observed power laws in distributions of change sizes as well as in time series of change. Also, we suggested a qualitative explanation of software evolution dynamics based on Bak's sandpile model.

8 Conclusion

We presented empirical evidence for the existence of fractal structures in the evolution of open source software systems. Fractal structures are identified and measured as power laws in space (across the system) and in time (over the system lifetime). Specifically, our 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 suggest that open source software systems follow the SOC dynamics during their evolution.

It is interesting that power laws have been observed at three levels of abstraction in terms of open source software development: the project level [HJ02][Koc04][MFT02], the file level (our work), and the level of lines of code [GY03]. These power laws make it promising to explain the evolution of open source communities as well as the evolution of individual projects using SOC. A unified framework built on SOC may be constructed and then enhanced in the future, within which software evolution may be better explained and understood.

References

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

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

[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.

[CHW05] Stephen Cook, Rachel Harrison, and Paul Wernick. A simulation model of self-organising evolvability in software systems. In *Proceedings of the IEEE International Workshop on Software Evolvability*, pages 17–22, Budapest, Hungary, September 2005.

[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.

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

[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.

[Fre04] FreeBSD. Website: http://www.freebsd.org, 2004.

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

[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.

[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.

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

[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.

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

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

[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.

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

[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.

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

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

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

[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.

[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.

[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.

[Net02] NetBSD. Website: http://www.netbsd.org, 2002.

[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.

[Ope04a] OpenBSD. Website: http://www.openbsd.org, 2004.

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

[Ope05] Open Source Definition. Website: http://www.opensource.org/docs/definition.php, 2005.

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

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

[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.

[Sch90] C. H. Scholz. *The Mechanics of Earthquakes and Faulting.* Cambridge University Press, 1990.

[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.

[Sou05] SourceForge. Website: http://sourceforge.net/, 2005.

[Spr03] J. C. Sprott. *Chaos and Time-Series Analysis.* Oxford University Press, 2003.

[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.

[WH06] Jingwei Wu and Richard C. Holt. An extractor suite for C and C++: Choosing the right tool for the job. 2006. Draft.

[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.

[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.

A Studied Open Source Software Systems

1 **BSD** is the UNIX derivative distributed by the University of California at Berkeley. The development of BSD at Berkeley ceased after 4.4BSD-Lite was delivered in 1995. Since then, several distributions based on 4.4BSD-Lite such as FreeBSD, NetBSD and OpenBSD have been in active development.
- **FreeBSD** is developed as a complete operating system with the kernel and miscellaneous utilities included [Fre04].
- **NetBSD** is a unified, portable, production-quality operating system [Net02]. It is often used in embedded systems.
- **OpenBSD** focuses on open source and documentation, standardization, code correctness and security [Ope04a].

2 **GCC** [GCC02] stands for GNU Compiler Collection and it has a set of 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.

3 **KOffice** [KOf04] is an integrated office suite developed for the K Desktop Environment (KDE). KOffice is part of the KDE Project and consists of 12 main applications: KPresenter, KWord, KChart, KSpread, 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.

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

5 **Linux** [Lin04] is a cloned operating system UNIX, written from scratch by Linus Torvalds and then subsequently worked on by hundreds of developers who are loosely connected through the Internet. It aims towards POSIX and Single UNIX Specification compliance. Linux 1.0 was delivered in March 1994.

6 **OpenSSL** [Ope04b] is a cryptography toolkit implementing the Secure Sockets Layer (SSL 2.0/3.0) and Transport Layer Security (TLS 1.0) protocols as well as a full-strength general purpose cryptography library.

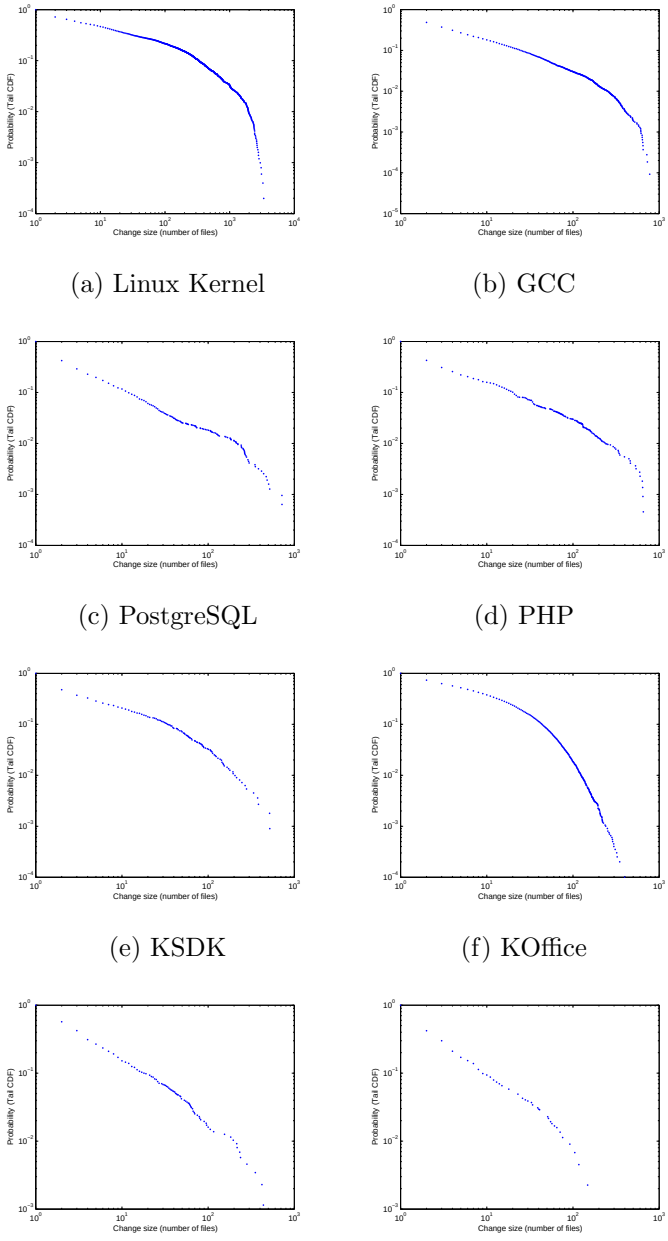
7 **PostgreSQL** [Pos03] is a SQL-compliant object-relational database management system (DBMS). It has more than 15 years history and a globally distributed development team.

8 **PHP** [PHP04], short for "Hypertext Preprocessor", is a reflective programming language which is widely used for developing server-side applications and dynamic web content.

9 **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.

B Distributions of Change Sizes



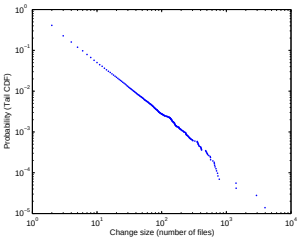
(a) Linux Kernel (b) GCC

(c) PostgreSQL (d) PHP

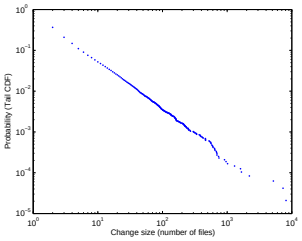
(e) KSDK (f) KOffice

(g) OpenSSL (h) Ruby

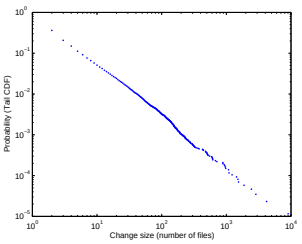
Fig. B.1. Tail CDF of Structural Changes



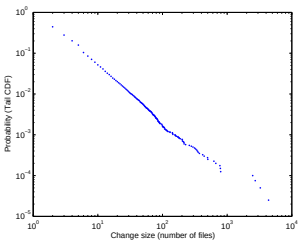
(a) FreeBSD



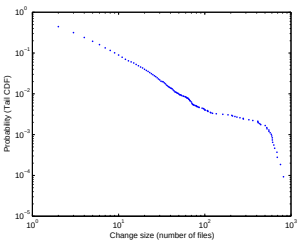
(b) OpenBSD



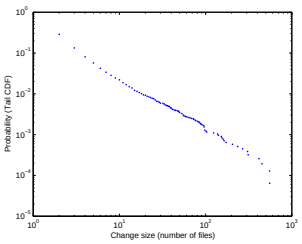
(c) NetBSD



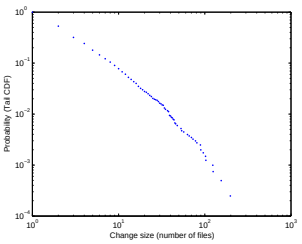
(d) GCC



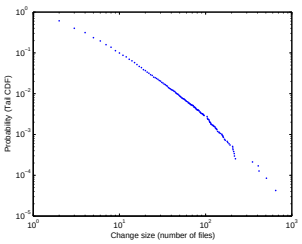
(e) PostgreSQL



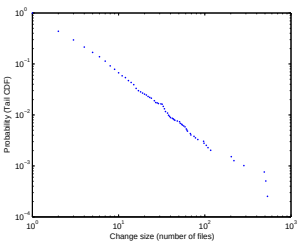
(f) PHP



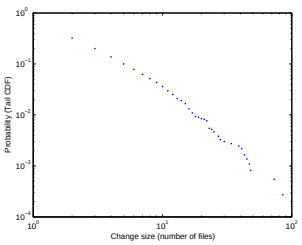
(g) KSDK



(h) KOffice



(i) OpenSSL



(j) Ruby

Fig. B.2. Tail CDF of Logical Changes