Evolution Spectrographs: Visualizing Punctuated Change in Software Evolution

Jingwei Wu, Claus W. Spitzer, Ahmed E. Hassan, Richard C. Holt
School of Computer Science
University of Waterloo
Waterloo, Canada
{j25wu, aeehassa, holt}@swag.uwaterloo.ca
cwspitzer@acm.org

Abstract

Software evolution is commonly characterized as a slow process of incremental change. Researchers have observed that software systems also exhibit characteristics of punctuation (sudden and discontinuous change) during their evolution. In this paper, we analyze punctuated evolution from the perspective of structural change. We developed a color-coded visualization technique called the Evolution Spectrograph (ESG). ESG can be applied to highlight conspicuous changes across a historical sequence of software releases. We describe evolution spectrographs and present the empirical results from our studies of three open source software systems: OpenSSH, PostgreSQL, and Linux. We show that punctuated change occurred in the evolution of these three systems, and we validate our empirical results by examining related software documents such as change logs and release notes.

1 Introduction

Software evolution is commonly characterized as a slow process of incremental change. This view is similar to Darwin's theory of *gradualism* on biological evolution [9]. According to Darwin, species develop slowly by means of a sequence of small mutations, and are gradually shaped by environmental selection into novel forms. Darwin's theory has been challenged in the field of biology. Eldredge and Gould proposed to view biological evolution as *punctuated equilibrium* [10]. 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 environmental selection. Researchers have pointed out discontinuous changes in software evolution [3, 4, 17, 20]. In this paper, we investigate whether software evolution is punctuated by means of studying dependency changes in software systems.

The rest of this paper is organized as follows. Section 2 presents our concept of punctuated software evolution. Section 3 describes a metrics-based framework for evolutionary analysis and a color-coded visualization technique called the Evolution Spectrograph. Section 4 discusses the empirical results from our studies of the evolution of open source software systems. Section 5 considers related work. Section 6 concludes this paper.

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 incremental change, and periods of punctuation, in which these underlying structures are fundamentally altered [14]. There are three main components of the theory: *deep structure*, *equilibrium*, and *punctuation*. We now give our interpretation of these three components in the context of software system evolution.

2.1 Deep Structure

According to Gersick, deep structure is a network of fundamental interdependent choices of the basic configuration into which a system's units are organized [14]. From this definition, we see similarities shared by deep structure and 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 [18]. 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 [13]. Therefore, we view software architecture as deep structure, which controls transitions between equilibrium and punctuation.

2.2 Punctuation Periods

Software architecture is the primary focus of change during punctuation periods. Qualitative change is made within a relatively compact period. By qualitative, we mean that change is made to alter the architecture in order to achieve stability in the long run. By compact, we mean that change is intense and occurs in short time.

As a software system evolves, its structure must be regularly adapted to changing requirements and environments else it becomes progressively less satisfactory [21]. At one extreme, an architecture may be altered fundamentally. For example, an application may be transformed from a centralized architecture to a distributed architecture. At the other extreme, an architecture may be partially adapted to new uses. For example, a single platform system may be restructured to support multiple platforms by introducing a virtual operating system service layer.

2.3 Equilibrium Periods

Within equilibrium periods, a system's architecture stays relatively stable, and it remains capable of accommodating forecasted changes in requirements. Changes are small and incremental. They rarely violate the principles imposed by the architecture. Otherwise, the architecture will deteriorate abruptly, thus making the system difficult to maintain. However, the architecture may exhibit symptoms of gradual decay, which are normally caused by the accumulating effects of maintenance activities, such as bug fixes and feature modifications.

Perry and Wolf observed two architectural decay phenomena: erosion and drift [25]. 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 a lack of coherence and clarity of form. The architecture needs to be evolved to counter erosion and drift. There are two strategies: (1) perform gradual change such as corrections and cleanups, and (2) resort to punctuated change to alter the architecture.

3 Methodology

In our studies of punctuated evolution, we have conjectured that punctuated change can be observed by examining changes to dependencies at the file level and functional growth measured in the number of files. There are several reasons for such an conjecture. First, software architecture is the highest-level abstraction of software systems, and it is difficult to extract. Creating subsystem hierarchies from scratch would be biased toward our understanding of what
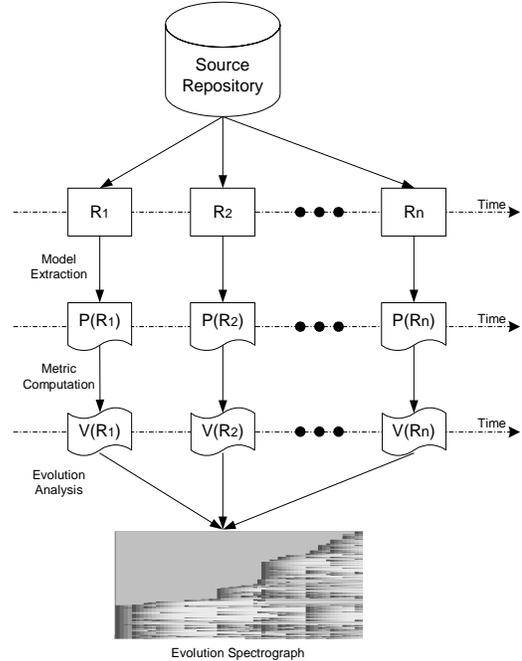

Figure 1. Framework for Studying Evolution

the architecture ought to be. Second, it not unusual that a software system (e.g., OpenSSH [24]) has a large number of files contained in only one or two directories. In this case, the directory structure cannot be adopted as the subsystem hierarchy. Third, functional growth is normally indicated by the growth of the number of files or modules [22]. Therefore, we have attempted to discover punctuated evolution at the file level.

We now describe a general metrics-based framework for evolutionary analysis and an innovative visualization technique called the Evolution Spectrograph (ESG), which provides visual cues to help identify punctuated change.

3.1 Analysis Framework

Our framework for analyzing software system evolution is based on three steps as shown in Figure 1.

Step 1: Model Extraction

We extract program models from a historical sequence of snapshots (builds or releases) of a software system. A large number of source code extractors are available to use, such as CCia [2], CAN [11], CPPX [8], and TkSeeSN [28]. The extractor we used is LDX [29], which is an instrumented version of the GNU code linker LD. It outputs function calls and variable uses. LDX is sufficiently robust and efficient to allow us to use it to automate fact extraction over hundreds of software releases.

In our studies, we chose to concentrate on file dependencies. We abstracted lower-level relationships to dependencies at the file level. In other words, we modelled a software system as a graph of files and dependencies among these files. The file dependency was not weighted, e.g., if several function calls occur from file F1 to file F2, we represent this as a single edge from F1 to F2.

Step 2: Metric Computation

In this step, metrics are computed from extracted program models $P(R_i)$. Depending on the extraction tools we could base our study of evolution on, various metrics can be computed on a per-file basis, for example, Lines of Code (LOC), Number of Functions (NOF), Cyclomatic Complexity, and Coupling/Cohesion. Metric computation can be performed for each source code release or for each development period (e.g., month). In Figure 1, metrics computed for all files in release R_i are stored in a vector denoted as $V(R_i)$.

In our studies, we chose to measure dependency change based on either incoming or outgoing dependencies. For example, if a file has one old incoming dependency removed and two new incoming dependencies added during a particular development period, we give it a metric value of 3 to denote its incoming dependency change in that period.

Step 3: Evolution Analysis

In this step, metrics can be aggregated, plotted and/or color-coded to portray the system's evolution. For example, one of the most important evolution analyses is to examine the system growth curve [15, 21]. We studied the development history of software systems using evolution spectrographs, which we will describe in the following section.

3.2 Evolution Spectrographs

A software evolution spectrograph is a color-coded chart that is used to display a metrics-based representation of the development history of a software system. We now describe how a spectrograph can highlight system growth and dependency change in one chart.

System Growth

System size is an important measure in the study of software evolution. Among the mostly widely used measures of size are the number of modules, the number of lines of code, and the number of functional points. Lehman suggests using the number of modules to measure the size of a large software system [22]. We adopted the number of files as the measurement of system size in our studies.

Dependency Change

Given any two releases of a software system, R_x and R_y, we use $P(R_x)$ and $P(R_y)$ to represent their extracted program models respectively. We define the change Δ_{xy} between R_x and R_y as follows:

$$U_{xy} = P(R_x) \cup P(R_y)$$
$$I_{xy} = P(R_x) \cap P(R_y)$$
$$\Delta_{xy} = U_{xy} - I_{xy}$$

If R_x is an earlier release than R_y, Δ_{xy} represents change required to evolve R_x to R_y. An extracted program model, in the simplest case, can be an ordered set (sequence) of lines of source code. A *diff* tool can be applied to determine what is added or deleted for any two software releases. In our studies, we were interested in analyzing how dependencies at the file level change over time. In this case, P can be seen as a graph with nodes representing files and edges representing static dependencies among these files. Therefore, Δ_{xy} can be seen as a dependency graph with no dependencies common to $P(R_x)$ and $P(R_y)$.

Evolution Matrix

An evolution matrix is a metrics-based representation that characterizes a software system's development history. If we adopt files as the basic unit for metric computation, the matrix comprises the metrics of all the files during the lifetime of the system. We will use the X axis to represent time and the Y axis to indicate the dimension of files. The matrix can be created based on either releases or development periods. In the release-based matrix, metrics are computed against program models $P(R_i)$. By contrast, in the period-based matrix, metrics are computed against change models Δ_{ij} that correspond to development periods bounded by R_i and R_j. In this paper, we will only discuss the period-based evolution matrix.

To derive a period-based evolution matrix, we divide the development process of a software system into a succession of fixed length periods, for example, on a monthly basis. If there are n files that are created during m periods, the matrix M has the dimension of $n \times m$. In this matrix, a row stores a vector of metrics that represents the change history of a file during the lifetime of the system. A column stores change metrics for all the files during a particular period.

Suppose we have a program, which has been developed for four months and consists of files A, B, C and D (see Figure 2). If we were to examine its change history based on outgoing dependencies, we first need to extract file dependency graphs for five snapshots of this system, R_i, where i is from 0 to 4. We then derive change graphs $\Delta_{i(i+1)}$ for each month. For each change graph , we compute the number of changed outgoing dependencies for each file. For
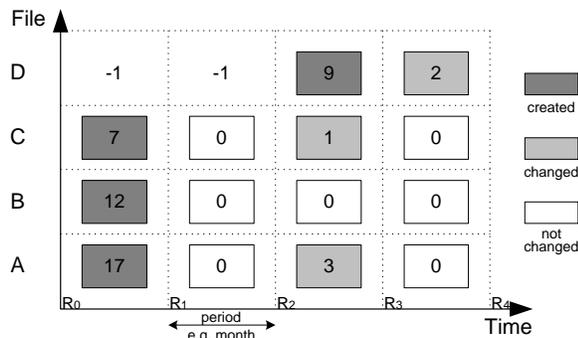
Figure 2. An Evolution Matrix

example, file A has 17 outgoing dependencies in the first month, and it is not changed in the second month, and it has 1 old dependencies deleted and 2 new dependencies added in the third month, and it is not changed in the last month. In this case, we describe dependency changes made to file A using a vector of metrics, (17, 0, 3, 0). If a file has not yet been created in a month, we give it -1 for that month. If it has been deleted, we also give it -1.

Color Chart

Once we have computed an evolution matrix M such as the one shown in Figure 2, we render it using a color chart C, in which the color of each cell is determined by its associated value and two colors, C_P (paint) and C_{BG} (background). If a file has not been created yet or has been deleted, we assign it color C_{BG}. If it is created or changed, we use color C_P to paint its corresponding cells. Otherwise, we determine its color using a color function g. We use $C[i][j]$ to denote the color of the cell located at row i and column j. We use $M[i][j]$ to denote the corresponding metric value. The color chart C is rendered according to the following equation.

$$C[i][j] = \begin{cases} C_{BG} & \text{if } M[i][j] = \text{-1} \\ C_P & \text{else if } M[i][i] \geqslant 1 \\ C_P & \text{else if } M[i][j\text{-1}] = \text{-1} \\ g\big(C[i][j\text{-1}]\big) & \text{otherwise} \end{cases}$$

where function g converts a darker color into a lighter one by decreasing saturation and increasing brightness. Given a color c, we use $h(c)$, $s(c)$ and $b(c)$ to represent its *hue*, *saturation* and *brightness* respectively. Thus, color c is expressed in the form of $[h(c), s(c), b(c)]$. We define function g as follows:

$$g(c) = [h(c), s(c) \times 0.8, b(c) \times 0.8 + 0.2]$$

After a file, F, is changed, its color becomes lighter and lighter as long as there is no change made to F. This gradient coloring process indicates that changes made to F become progressively less important as the system evolves. In other words, F starts to cool down if no future change occurs to it. If there are system-wide changes during a particular development period, all changed files will be painted in color C_P. A notable vertical band will appear during that period. It indicates a system-wide evolutionary event.

Figure 3 shows a color chart for the example system. We see two dark-green vertical bands appearing in the first and third months respectively. It means the system went through major structural change. The color of file B changes from dark-green to light-green in the form of gradient change. It means that file B has not been modified for a long period. By contrast, if a file's color has been constantly dark (e.g., file D), it means this file is being changed repeatedly.
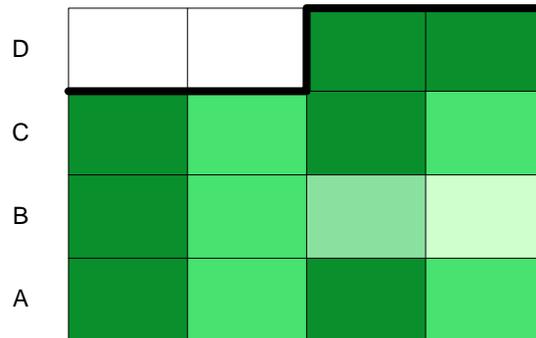

Figure 3. A Color Chart

File Ordering

Files are sorted according to the date of their creation. Such an order ensures that file F1 appears before file F2 if F1 was created earlier than F2. The order of files in Figure 3 is A→B→C→D. At the top of this figure, we can see a thick black line below the unshaded area. This line represents the system's growth. The system size is 3 files in the first two months and 4 files in the last two months. By preserving this order, we can investigate the correlation between new functionality and changes within the system. We will discuss this in more detail in Section 4.

Because the example system is small and has only a short lifetime of four months, we are unable to show its growth curve clearly. Also, there are no conspicuous vertical bands in the given spectrograph. In Section 4, we will present several evolution spectrographs for open source software systems.
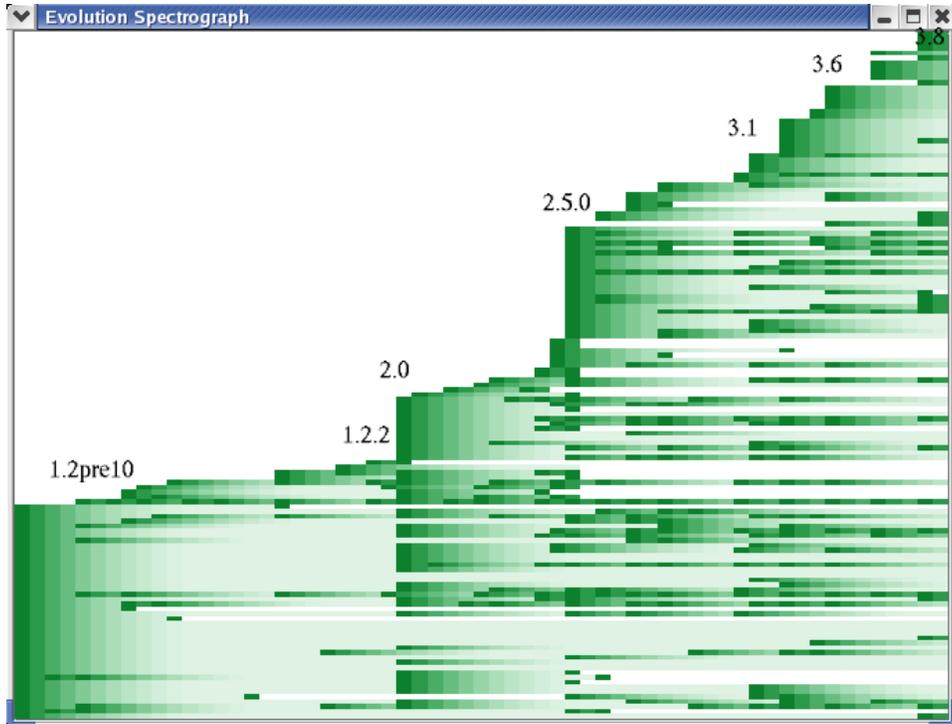
4

(a) The Spectrograph for Incoming Dependency Change
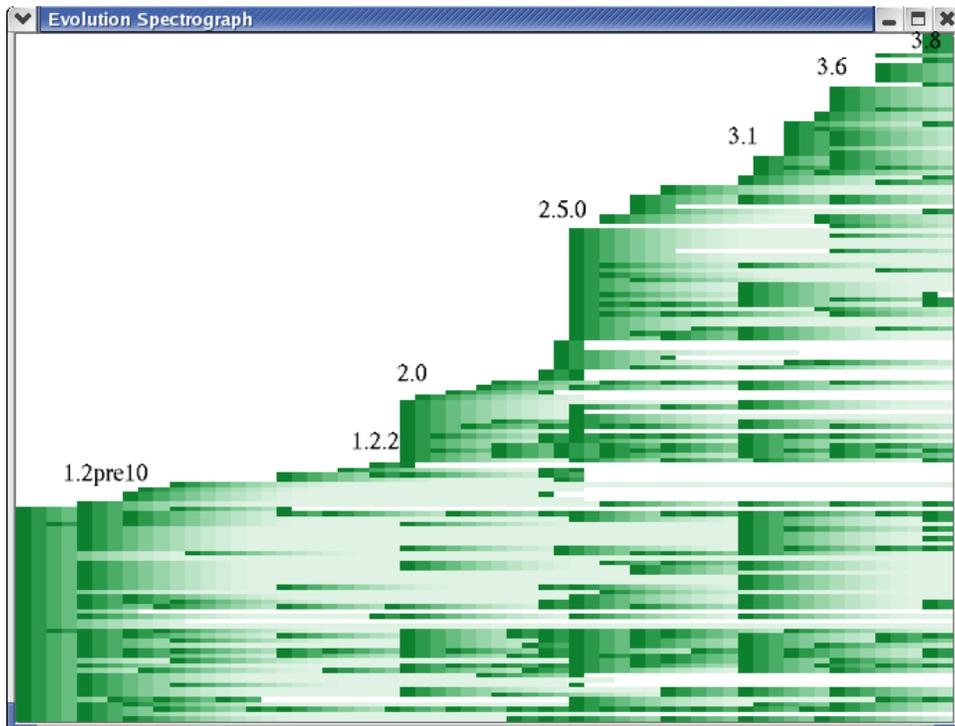
(b) The Spectrograph for Outgoing Dependency Change

Figure 4. The Evolution Spectrographs of OpenSSH

Application	Type	Language	Periods Studied	Size (KLOC)	# Releases
OpenSSH	Protocol	C	Oct 1999 – Mar 2004	22 – 70	60
PostgreSQL	DBMS	C	Jan 1997 – Dec 2003	185 – 525	85
Linux kernel	OS	C	Jun 1996 – Jul 2003	674 – 5141	324

Table 1. Properties of the Studied Systems

4 Case Studies

We looked for evidence of punctuated change by visualizing the evolution of three open source software systems, OpenSSH, PostgreSQL and Linux. We relied on evolution spectrographs to highlight conspicuous changes to both incoming and outgoing dependencies that occurred during the lifetime of these systems. This section presents the results of our work.

Table 1 summarizes the properties of the three systems. Their sizes are dramatically different from one another by an order of magnitude. The smallest of the three systems is OpenSSH, which has grown from 22 KLOC to 70 KLOC. The largest is the Linux kernel, which is more than five million lines of code now.

4.1 OpenSSH

OpenSSH is a well-known open source implementation of the Secure Shell (SSH) protocol suite of network connectivity tools [24]. OpenSSH encrypts communication traffic in order to effectively eliminate eavesdropping, connection hijacking, and other network attacks. In our case studies, we examined 60 releases since its initial release in October 1999.

Figure 4 presents two ESG charts of OpenSSH, in which changes to dependencies at the file level are plotted against release numbers. We label major release numbers so we can correlate them with conspicuous change events in the form of vertical bands (dark colored). We notice that changes made 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. It indicates that changes made to OpenSSH were not incremental. They occurred in system-wide bursts instead.

We speculate 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 in Figure 4. To validate our speculation, we examined the related online documentation. In May 2000, the developers of OpenSSH implemented support for the SSH2 protocol and release 2.0 was delivered. We map this major change to the first punctuation. Given that OpenSSH is a protocol-centered application, it is not surprising that this protocol update resulted in system-wide changes. In November 2000, the developers implemented Secure File Transfer Protocol (SFTP) client and server support, which was shipped with release 2.5.0 in February 2001. We associate this functionality enhancement with the second punctuation. We are currently unable to identify what caused the third punctuation near release 3.1 based on the available documentation. It remains to be investigated in future work.

We attribute new functionality as the main driving force of change during the two punctuation periods near releases 2.0 and 2.5.0. For each of these periods, the system size, which was measured using the number of files in Figure 4, showed a substantial growth. By contrast, the system did not grow much during the third punctuation period, but the internal system structure went through substantial change. We relate the third punctuation to the transition from release 2 to release 3. We believe that the architecture of OpenSSH was reworked during that time in order to support future evolution.

We gain insights into the causes of conspicuous change events by means of correlating ESG charts for both incoming and outgoing dependency changes. For example, if we only look at Figure 4(b), it seems that release 1.2pre10 was aggressively restructured. However, Figure 4(a) shows this restructuring was actually caused by adding or eliminating dependencies to seven files, which are indicated by seven horizontal lines in a relatively dark color. A further examination of the source code revealed that the logging utilities `log-client.c` and `log-server.c` were merged into a new file called `log.c`. This merge resulted in widespread changes in the extracted dependency graph but not in the source code. It is interesting that this phenomenon can be explained as a change to a single "aspect", namely, logging in the system.

4.2 PostgreSQL

PostgreSQL is a large Object-Relational Database Management System (DBMS) [26]. Its development started in 1996 at the University of California at Berkeley, and it soon became an open source project with a globally distributed development team. We analyzed monthly builds rather than official releases in order to perform a longitudinal analysis of monthly change. We extracted 85 monthly builds, which cover in total 84 months of development from January 01,
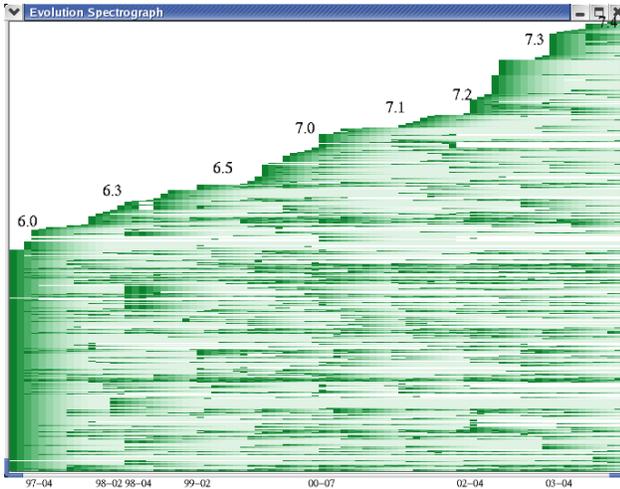
(a) The Spectrograph for Incoming Dependency Change
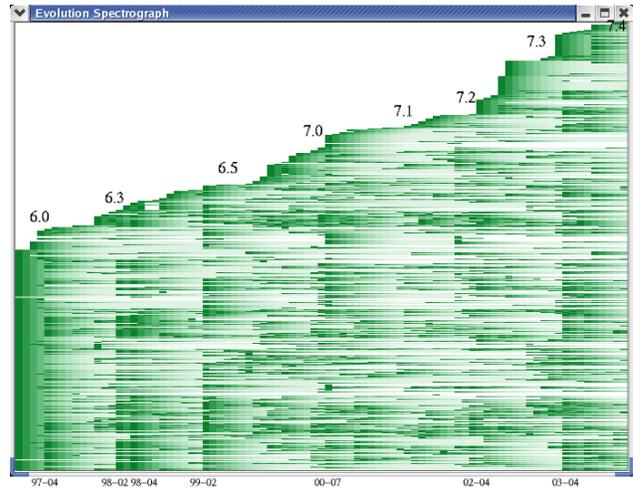
(b) The Spectrograph for Outgoing Dependency Change

Figure 5. The Evolution Spectrographs of PostgreSQL

1997 to January 01, 2004.

Figure 5 shows two spectrographs of PostgreSQL. The pictures indicate that the system growth of PostgreSQL is approximately linear in terms of the number of files. We notice that conspicuous vertical bands appear in Figure 5(b) but not in Figure 5(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 indicates that each system-wide perturbation highlighted by a vertical band in Figure 5(b), was actually related to a small number of files that had their incoming dependencies substantially changed.

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 2. Percentage of Changed Files

We calculated the percentage of changed files for seven months, which correspond to the vertical bands in Figure 5(b). We examined two kinds of changes: (A) files are considered to be changed if their dependencies are changed; (B) files are considered to be changed if their dependencies

to non-`utils`[1] subsystems are changed. Table 2 summarizes the calculated results. Both July 2000 and April 2002 have very high change rates in both cases. By contrast, the other five months have much lower change rates in case B as opposed to case A. We further examined what file dependencies were changed in these five months. We found that the incoming dependencies of the services provided by the `utils` subsystem accounted for the majority of changed dependencies. These services include error reporting and logging (`error`), memory management (`mmgr`), and cache utility (`cache`). This is a very interesting phenomenon. It shows the developers of PostgreSQL have continually devoted efforts to several quality attributes such as reliability and performance. These quality attributes are important factors to the success of a database management application in highly competitive market.

We conclude that PostgreSQL's evolution is punctuated based on the above examination. We also notice that punctuations are evenly distributed during the system's lifetime. Recalling the system grew approximately at a linear rate, we believe 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 [23]. Any new features first exist as patches for a relatively long time, and are

---

[1] 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.

then incorporated into the core version only after they are approved by the core members.

4.3 The 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 [1]. The first official release, version 1.0, occurred 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).

We analyzed the evolution of Linux, from 2.0 to 2.5.75. Our study involved 324 releases, which cover 7 years of development. These releases were ordered according to their release dates. We created two spectrographs to characterize how the kernel evolved on a monthly basis. The vertical bands in Figure 6 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 were no official kernel releases from June 2000 to December 2000, this punctuation was apparently caused by the accumulating effects of six months of development. In addition, there was no considerable system growth in release 2.4.0. This is not normal in Linux's evolution given the fact that Linux has been growing exponentially [15, 29]. It shows that Linux experienced critical structural change before the milestone release 2.4.0.

The other vertical bands are related to new functionality added to the kernel, and they indicate three periods of functional punctuation. The first functional punctuation spanned about three months, covering development versions 2.1.24 – 2.1.40. These versions were enhanced for multi-processor support and tuned for faster speed [16]. The second functional punctuation involved development versions 2.3.20 – 2.3.35. The three major subsystems `fs`, `net` and `kernel` experienced substantial change. For example, several Journaling File Systems such as XFS from SGI and JFS from IBM were introduced; the network subsystem (`net`) was split into two layers: network address translation and packet filtering; the use of the global Big Kernel Lock (BKL) was minimized by replacing it with local subsystem spinlocks; and Symmetric Multiprocessor Processing (SMP) was enhanced to support up to 32 CPUs. The third functional punctuation occurred with versions 2.5.15 – 2.5.40. It was aimed at supporting high end enterprise servers and embedded devices [1, 27]. 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.

5 Related Work

Lehman's work on OS-360 revealed that the growing entropy of local changes was counteracted by periodic global effort [20]. Lehman found that the cost and unreliability of software changes in OS-360 rose sharply during a series of minor releases and then restabilized when the architecture was readapted every few major releases. His studies were mainly based on system growth. By contrast, we measured punctuated evolution in open source software in terms of structural dependency change.

Antón and Potts adopted a punctuated equilibrium model to study the evolution of telephone systems [3]. Their study was based on the longitudinal analysis of feature growth in telephone systems. They observed that a burst of telephone features occurred every 8 to 12 years. By contrast, we relied on spectrographs of structural change to locate punctuated evolution. We observed more frequent occurrences of punctuated change, namely, every 1 or 2 years. This perhaps is because our analysis is at a lower level granularity of change as opposed to their feature analysis. In addition, the three software systems we analyzed are much younger than the telephone system they investigated.

Aoyama observed that discontinuous evolution occurred in mobile phone software systems across multiple product lines [4]. 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 computing-centric style. The author studied the system growth and the conceptual architecture. By contrast, our studies were based on the visualization of structural change at the file level. Another difference is that our case systems are open source software.

Barry et al. have shown that different software systems may share the same lifetime evolution pattern, which can be determined using a software volatility measurement with three dimensions: amplitude, periodicity and dispersion [6]. Their approach computes a metric value for each release or phase, while our approach computes an evolution matrix to characterize software system evolution across all releases.

We now consider related work in the field of software evolution visualization. Gall and Jazayeri used *percentage bars* to show three kinds of evolutionary entities (structure, attribute, and time) simultaneously in one view [12]. Their aim was to find conspicuous changes from the history of software releases to assist in future restructuring. We used spectrographs to combine system growth, system change and release history simultaneously. Our goal was to gain a better understanding of the paradigm of software evolution by looking for evidence of punctuated change.
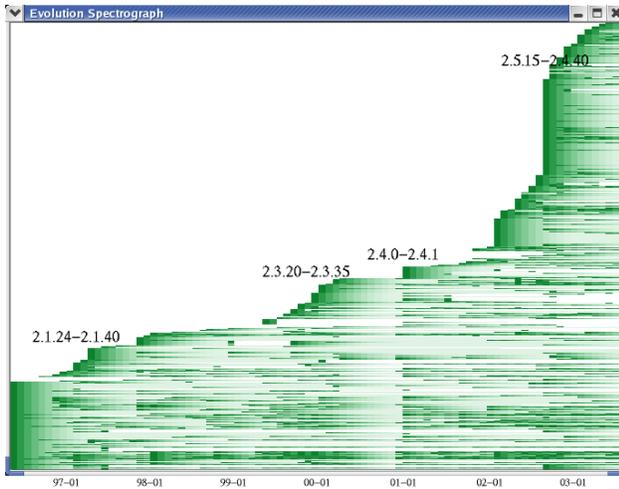
(a) The Spectrograph for Incoming Dependency Change
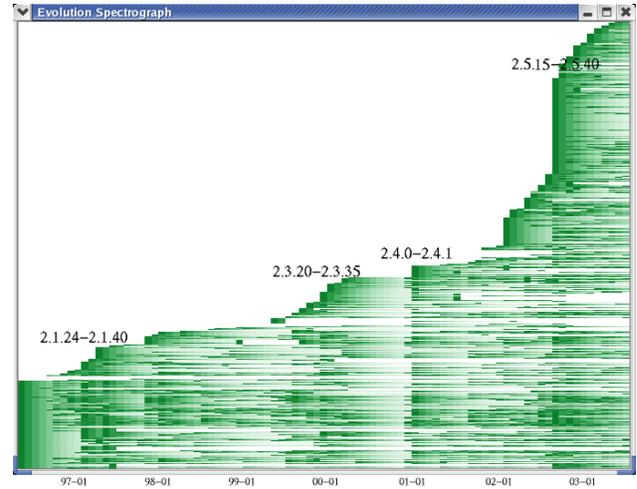
(b) The Spectrograph for Outgoing Dependency Change

Figure 6. The Evolution Spectrographs of Linux

Lanza proposed *Evolution Matrix* as a means to recover the evolution of object oriented software systems [19]. In his evolution matrix, 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 change patterns over time. By contrast, we only rely on gradient coloring to emphasize visual cues for punctuated change. In addition, we sort files according to their creation dates so that we can correlate functional growth with punctuated change.

Eick et al. have developed a set of tools for visualizing several data classes, such as code version history and release differences [5]. They represented lines of code using the color-coded pixels in order to achieve a higher information density. Their approach encodes the third dimension for the history of releases into color pixels. This makes it not appropriate for finding punctuated change from a historical sequence of releases.

Collberg et al. have developed a graph-based system for visualizing software evolution, called GEVOL [7]. The evolution history of software is characterized using a sequence of graphs, each of which represents the state of the system at a given point in time. GEVOL uses advanced layout algorithms in order to preserve the viewer's mental map. Colors are applied to indicate change over time. GEVOL is useful for visualizing how software structures evolve, such as call graphs, control graphs, and inheritance graphs. Our approach can be customized to show how measured software properties change over time.

6 Conclusions and Future Work

We presented a visualization technique called Evolution Spectrographs. A spectrograph combines metrics and gradient colors to portray software system evolution and to highlight main evolutionary events. The system growth curve is displayed as a part of the spectrograph when the files on the axis Y are sorted by creation date. This visualization technique allows the viewer to perceive punctuated change with relative ease.

We have used evolution spectrographs in the analysis of three open source software systems, OpenSSH, PostgreSQL and Linux. Our work has shown that these three systems evolved in punctuations.

Below, we describe a list of possible future work:

- Use of subsystem hierarchies. An architecture is normally organized as a hierarchical decomposition based on subsystems. By exploring the subsystem hierarchy, we can distinguish higher-level dependency changes from lower-level dependency changes. Therefore, we can investigate punctuated change at varying levels of granularity.

- Exploration of other metrics. We measured changes to structural dependencies in this paper. It is interesting to try other metrics in order to gain insights into software system evolution. For example, we could include metrics such as defect density.

- Comparison of applications in the same domain. The

9

three systems we examined are targeted at dramatically different domains. We would like to examine similar applications and compare them. For example, MySQL and PostgreSQL are two ideal DBMS candidates for comparison.

- Identification of aspects. As discussed in section 4.1, changes related to software aspects (e.g., logging) often result in conspicuous events in an evolution spectrograph. By choosing appropriate measurements and filtering noises, we may use evolution spectrographs to help identify important aspects in an software system and to facilitate system maintenance tasks.

Acknowledgements

References

[1] *The Linux Kernel Archives*. http://www.kernel.org, 2003.

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

[3] A. I. Antón and C. 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.

[4] M. Aoyama. Continuous and discontinuous software evolution: Aspects of software evolution across multiple produce lines. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 87–90, Vienna, Austria, September 10-11 2001. ACM Press.

[5] T. Ball and S. G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, April 1996.

[6] E. J. Barry, C. F. Kemerer, and S. 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 03-10 2003.

[7] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 77–86, San Diego, California, June 11-13 2003.

[8] CPPX. *The C/C++ Source Extractor Online References*. http://swag.uwaterloo.ca/~cppx, 2002.

[9] C. Darwin. *The Origin of Species by Means of Natural Selection*. 1859.

[10] N. Eldredge and S. Gould. *Models of Paleobiology*, 1972.

[11] R. Ferenc, A. Beszedes, F. Magyar, and T. Gyimothy. A short introduction to columbus/can. *Technical Report*, 2001.

[12] H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: The use of color and third dimension. In *Proceedings of the International Conference on Software Maintenance*, pages 99–108, Oxford, England, August 30-September 3 1999.

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

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

[15] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance*, pages 131–142, San Jose, California, October 11-14 2000.

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

[17] A. A. Gorshenev and Yu. M. Pis'mak. Punctuated equilibrium in software evolution. *DOI: cond-mat/0307201*, July 09 2003.

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

[19] M. 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 10-11 2001. ACM Press.

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

[21] 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.

[22] 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.

[23] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. 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, USA, May 19-20 2002. ACM Press.

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

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

[26] PostgreSQL. *The PostgreSQL Database System Online References*. http://www.postgresql.org, 2003.

[27] A. K. Santhanam. *Towards Linux 2.6: A Look into the Next Kernel*. http://www.ibm.com/developerworks/linux/library, 2003.

[28] TkSee/SN. *The TkSee/SN Source Code Extractor Online References*. http://www.site.uottawa.ca/~tcl/kbre, 2003.

[29] J. Wu and R. 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*, Barcelona, Spain, March 28 2004.