# Towards a Better Understanding of Software Evolution: An Empirical Study on Open Source Software

Guowu Xie    Jianbo Chen    Iulian Neamtiu
Department of Computer Science and Engineering
University of California, Riverside
{xieg,jianbo,neamtiu}@cs.ucr.edu

## Abstract

*Software evolution is a fact of life. Over the past thirty years, researchers have proposed hypotheses on how software changes, and provided evidence that both supports and refutes these hypotheses. To paint a clearer image of the software evolution process, we performed an empirical study on long spans in the lifetime of seven open source projects. Our analysis covers 653 official releases, and a combined 69 years of evolution. We first tried to verify Lehman's laws of software evolution. Our findings indicate that several of these laws are confirmed, while the rest can be either confirmed or infirmed depending on the laws' operational definitions. Second, we analyze the growth rate for projects' development and maintenance branches, and the distribution of software changes. We find similarities in the evolution patterns of the programs we studied, which brings us closer to constructing rigorous models for software evolution.*

## 1. Introduction

Software continues to evolve long after the first version has shipped. Numerous estimates indicate that the costs associated with software maintenance and evolution are at least 50%, and sometimes more than 90% of total costs associated with a software system [6]. To reduce these costs, both managers and developers must understand the factors that drive software evolution and take proactive steps that facilitate changes and ensure software does not decay.

We now have access to the repositories of large open source applications with lifetimes that exceed 20 years. Our work leverages software evolution data contained in historic program versions, and tries to paint a clearer image of the software evolution process. To this end, we analyzed the complete release histories of Samba, Bind 9, OpenSSH, SQLite, and Vsftpd, as well as the past 15 years of Sendmail and the past 5 years of Quagga. In total, our study covers 653 official releases and over 69 years of cumulative program evolution.

In the first part of our paper, we try to verify whether existing software evolution models apply to our test programs. In particular, we are interested in Lehman's eight laws of software evolution. First formulated in the early 1970s, in Belady and Lehman's study on the evolution of OS/360 [1], these laws essentially characterize the software evolution process as a self-regulating and self-stabilizing system, subject to continuing growth and change [11, 8, 10]. The laws are named after traits of the software evolution process: "I - Continuing Change", "II - Increasing Complexity", "III - Self Regulation", "IV - Conservation of Organizational Stability", "V - Conservation of Familiarity", "VI - Continuing Growth", "VII - Declining Quality", and "VIII - Feedback System". We use source code metrics, as well as project and defect information to analyze software growth, characterize software changes, and assess software quality. The results of our study indicate that laws I, II, III, and VI are confirmed, while for the remaining laws—IV, V, VII, and VIII—we found evidence to the contrary, or a more precise operational definition is needed. We present details on our findings in Section 4. To our knowledge, ours is the first study to explicitly consider each of the eight laws, and test each law using a variety of measures, on long spans of program evolution. Moreover, we try to address a challenge mentioned by Lehman et al. [9], i.e., separating the characterizations of system growth and system change.

In the second part of the paper (Section 5), we present our own observations on how software evolves, based on similarities in the evolution patterns of the programs we studied. In particular, when analyzing both the development and maintenance branches for each application we found that the growth rate is super-linear on the main development branches and at most linear on maintenance branches. When analyzing program changes at a fine-grained level, we found that distribution of changes largely follows power laws, i.e., the majority of changes are concentrated to a



978-1-4244-4828-9/09/$25.00 2009 IEEE    Proc. ICSM 2009, Edmonton, Canada

small fraction of the source code. Finally, we found that changes to interfaces are on average an order of magnitude less frequent than changes to implementation.

The remainder of the paper presents an overview of the applications (Section 2) and the methodology we followed in our study (Section 3); the examination of Lehman's laws (Section 4), our observations (Section 5), and threats to validity (Section 6).

## 2. Applications

We ran our empirical study on seven open source applications written in C. In selecting our test applications, we used several criteria. First, since we are interested in long-term software evolution, the applications had to have a long release history. Second, applications had to be sizable, so we can understand the issues that appear in the evolution of realistic, multi-developer software. Third, the applications had to be actively maintained.

Table 1 presents high-level data on application evolution. The second and third columns present the time span we considered for each application and the number of official releases, respectively. The rest of the columns present information (version, date and size) for the first and last releases.

We aimed to analyze complete lifespans for each application, from the first publicly available release to the latest release as of March 2009. For two applications, however, Sendmail and Quagga, their initial versions are so old that we could not process them with our tools, e.g., preprocess or compile them, since they use antiquated headers, libraries, or even rely on old versions of Gcc.

We now provide an overview of each application.

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

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

**BIND** is the leading DNS server on the Internet. According to its official history (`https://www.isc.org/software/bind/history`), BIND development goes back to the early 1980s, but the current line, BIND 9, is a major rewrite. We analyzed all the BIND 9 versions, from 9.0.0b1 (Feb. 2000) to 9.6.1b1 (March 2009).

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

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

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

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

As we can see in Table 1, excepting Quagga, all programs have grown considerably relative to their initial versions.

## 3. Methodology

For each application, we followed the same procedure. We first downloaded all publicly available official releases, starting with the most recent one and going back as far as we could. We then configured and preprocessed the main server in each release, excluding test programs or various clients that ship with the server. Finally, we "merged" all the source code that goes into building the server into a single `.c` file, using the CIL merger tool [15], however retaining module information. This strategy ensured we focused on the evolution of one self-contained, standalone program. Note that the LOC numbers in Table 1 show the source code size for the server program we analyzed. The LOC numbers for the entire application (e.g., including clients or testing infrastructure) are certainly larger, but they do not constitute our focus and we do not present them here. We tried to keep the configuration (compiler flags, module options) consistent from version to version. For each version, we made sure we could compile, link, and run the server.

Finally, we ran two source code analysis tools, ASTdiff and RSM, to collect data on the server program's evolution. ASTdiff is a tool we developed that compares C programs by matching their abstract syntax trees. ASTdiff collects a variety of change metrics, e.g., changes to types, global variables, function signatures and bodies. While the core algorithm and some case studies are presented in our previous work [14], for this work we enhanced ASTdiff to support collecting information about code complexity (i.e., common coupling, function calls per function) and modules. RSM (Resource Standard Metrics [17]) is a commercial tool that we used for computing cyclomatic complexity.

| Program | Time frame (years) | Releases | First release | | | Last release | | |
|---|---|---|---|---|---|---|---|---|
| | | | Version | Date | Size (LOC) | Version | Date | Size (LOC) |
| Samba | 15 | 89 | 1.5.14 | 12/08/1993 | 5,514 | 3.3.1 | 02/24/2009 | 1,045,928 |
| Sendmail | 15 | 57 | 8.6.4 | 10/31/1993 | 25,912 | 8.14.4a | 01/13/2009 | 87,842 |
| Bind | 9 | 168 | 9.0.0b1 | 02/04/2000 | 169,306 | 9.6.1b1 | 03/12/2009 | 321,689 |
| OpenSSH | 9 | 78 | 1.0pre2 | 10/27/1999 | 12,819 | 5.2p1 | 02/22/2009 | 52,284 |
| SQLite | 8 | 172 | 1.0 | 08/17/2000 | 17,273 | 3.6.11 | 02/18/2009 | 65,108 |
| Vsftpd | 8 | 60 | 0.0.9 | 01/28/2001 | 6,774 | 2.1.0 | 01/21/2009 | 15,711 |
| Quagga | 5 | 29 | 0.96 | 08/12/2003 | 41,623 | 0.99.11 | 09/05/2008 | 47,511 |

Table 1: Application information.

## 4. Lehman's Laws of Software Evolution

The first part of our empirical study tries to verify each of Lehman's eight software evolution laws on our test applications. For each law, we describe the metrics we used and our observations on whether the law is confirmed, infirmed, or a more precise definition is needed.

### 4.1. Continuing Change

The first law postulates that a program must continually adapt to its environment, otherwise it becomes progressively less useful [8]. All our projects are widely used and actively maintained, so if the law holds, we should observe that programs are continually undergoing change. To characterize change, prior approaches have used the number of modules handled in each release [1, 7, 2], system and module size [9, 4, 3], function modifications and complexity [16]. We employ a variety of metrics. In Figure 1 we present the cumulative number of changes as well as the ratio of changes over more than 15 years for Samba. The top graph plots the cumulative number of changes to program elements, i.e., functions, types and global variables. The "modification" graph shows the cumulative number of changes to function bodies and signatures, type definitions, as well as changes to global variable types and definitions. The "addition" graph shows the cumulative number of function, types, and global variables added to the program. Finally, the "deletion" graph shows the cumulative number of function, types, and global variables deleted from the program. The bottom graph shows how changes are split among functions, types, and global variables, for each release. We found that the majority of changes are made to functions, a reason why other researchers only consider functions when presenting system change and growth [16, 2]. Due to space constraints, we only present these graphs for Samba, however, the trends are similar for the other programs.

We make several observations on how the seven programs have changed over time. First, the figure clearly shows that applications continue to change over time; in fact the total number of changes (not pictured) is the sum of the three graphs for each application. While the rate of change subsides for later versions, this only shows that change happens at a slower pace. Second, we observe that additions are more common than deletions, a factor that will help us test the "continuing growth" law later on, in Section 4.6. Third, changes to interfaces are much less frequent than changes to implementation, an aspect we will return to in Section 5.3.

Therefore, we conclude that Lehman's first law is confirmed for our test programs.

### 4.2. Increasing Complexity

The second law postulates that as a program evolves, its complexity increases, unless proactive measures are taken to reduce or stabilize the complexity [8].

In an early work by Lehman [1], complexity was defined as the percentage of modules handled relative to the total number of modules; Lawrence [7] uses this definition, as well as programmer productivity. Later work by Kemerer and Slaughter [5] suggests normalized cyclomatic complexity by LOC as a metric, Paulson et al. [16] use average function complexity, while Wu and Holt [20] employ metrics such as function calls per function and common coupling.

We believe large values for cyclomatic complexity, common coupling and function calls per function hinder evolution by making the program difficult to understand and difficult to change. Therefore, we measure complexity using the average number of function calls per function, McCabe's cyclomatic complexity, and common coupling. For the latter two metrics, we present both absolute and normalized values.

In Table 2 we present the results of running a linear regression where the independent variable is $D/365$ ($D$ is the number of days since the initial release), and the dependent variable is the value of the complexity metric. Regarding function calls per function, for those programs where

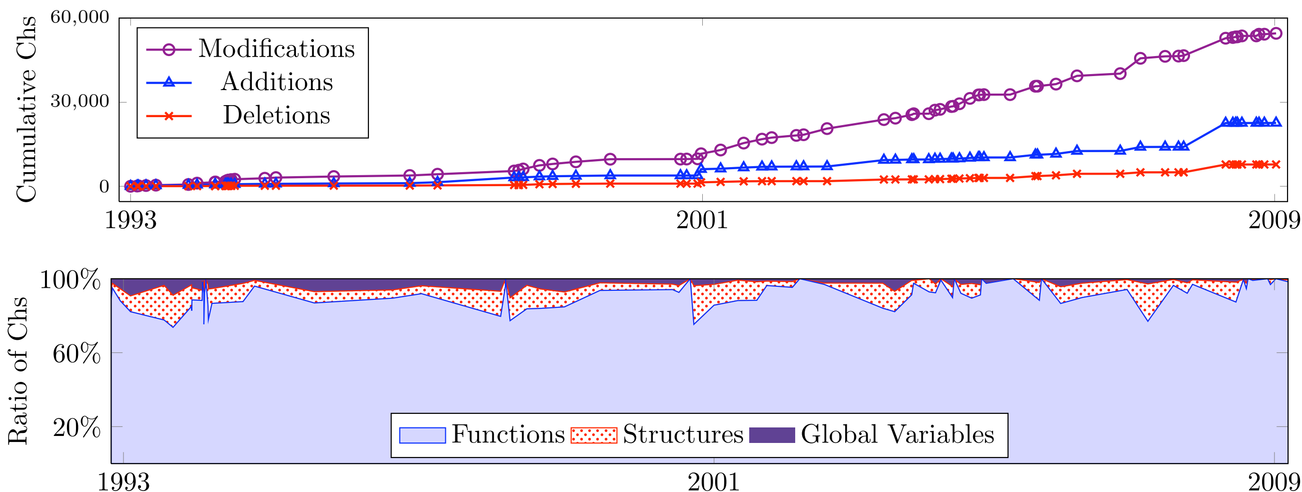


Figure 1: Cumulative changes and ratio of changes to Samba.

| Program | Function calls per function(avg.) | | Cyclomatic complexity | | | | Common coupling | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Total | | Normalized | | Total | | Normalized | |
| | $\beta$ | $R^2$ | $\beta$ | $R^2$ | $\beta$ | $R^2$ | $\beta$ | $R^2$ | $\beta$ | $R^2$ |
| Samba | 0.169 | 0.561 | 5732.869 | 0.864 | 0.910 | 0.316 | 162.519 | 0.912 | -0.001 | 0.221 |
| Sendmail | -0.666 | 0.332 | 2764.958 | 0.942 | 1.871 | 0.667 | 173.249 | 0.793 | -0.035 | 0.569 |
| Bind | 0.169 | 0.561 | 5732.869 | 0.864 | 0.910 | 0.316 | 162.519 | 0.912 | -0.001 | 0.221 |
| OpenSSH | -1.100 | 0.4341 | 1988.498 | 0.897 | 1.640 | 0.697 | 169.172 | 0.883 | -0.013 | 0.749 |
| SQLite | -1.643 | 0.891 | 2558.990 | 0.939 | 2.375 | 0.190 | 118.386 | 0.976 | -0.016 | 0.417 |
| Vsftpd | 0.641 | 0.826 | 432.903 | 0.863 | 4.051 | 0.794 | 24.880 | 0.814 | -0.029 | 0.814 |
| Quagga | -0.285 | 0.328 | 518.110 | 0.373 | -3.086 | 0.342 | 43.980 | 0.923 | 0.004 | 0.595 |

Table 2: Slope and correlation coefficients showing how program complexity changes over time.

the coefficient of correlation is high, e.g., Bind, SQLite and Vsftpd, we observe both negative and positive correlation, which suggests both decreasing and increasing trends. Unsurprisingly, we find the absolute values for cyclomatic complexity and common coupling to increase, since program size increases. However, when normalizing common coupling by the number of possible couplings between modules, $N(N-1)/2$, we can notice mostly negative trends, e.g., the $\beta$ values in column 10. We performed a preliminary analysis of changes between releases and found that complexity-reducing measures are rarely taken, hence the decrease in normalized complexity is due to increasing size, rather than decreasing complexity. Finally, we computed the average size of a module and found it to be slightly increasing, which makes software harder to maintain; we omit graphs due to lack of space.

Testing this law is difficult, as initially pointed out by Lawrence [7]. Even with commit or release notes at hand, it is hard to pinpoint those efforts specifically meant to reduce complexity. However, the increasing complexity trends we observed lead us to conclude that this law holds for the applications we examined.

### 4.3. Self Regulation

Lehman et al. [9] suggest that the evolution of large software systems is a self-regulating process, i.e., the system will adjust its size throughout its lifetime. This translates to observing "ripples"—small negative and positive adjustments— in the growth trend of a system. To verify this law, we analyzed the incremental module growth for each system. Due to lack of space, we only present

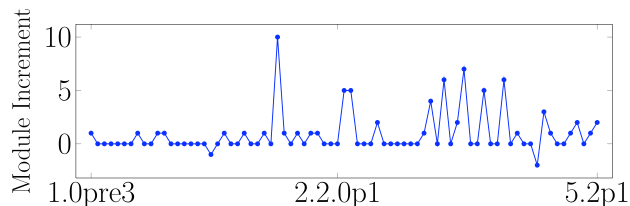


Figure 2: Incremental module growth for OpenSSH.

the graph for OpenSSH. Figure 2 shows the module increment on the Y axis, while the X axis is release number. We observe that these ripples exist indeed, and positive adjustments are more frequent than negative adjustments, a trend shared by the rest of the applications. The same behavior is observed when considering the number of functions as metric for system size. Therefore, we conclude that the law of self-regulation is confirmed for our test programs.

### 4.4. Conservation of Organizational Stability

This law, also known as "invariant work rate", stipulates that the rate of productive output tends to stay constant throughout a program's life time. Finding accurate metrics for work rate is difficult, since effort does not equate progress, especially for large projects where communication costs are high [9, 8]. Lehman et al. [9] suggest using the number of changes per release as possible work rate indicator, but leave this to future work. Taking a cue from their paper, we analyze the programs using two definitions for work rate: (1) the average number of changes per day, i.e., for each release $i$, we divide the total number of changes introduced in $i$ by the number of days between release $i-1$ and $i$ (which has the advantage of being invariant to release intervals), and (2) change and growth rates, i.e., the number of function additions and function changes divided by the total number of functions. We found that the work rate computed using metric 1 is not invariant. Using metric 2, we found that the change and growth rates do not subside (see Figure 3), which suggests larger efforts as programs grow.

Intuitively, these trends make sense since the programs are open source, and the number of developers tends to increase over a program's lifetime [12]. Since the invariant work rate law, in its original version, was formulated in the context of commercial software development with limited resources, here we do not have enough data to determine if this law is applicable to our examined open source programs or not.

### 4.5. Conservation of Familiarity

This law suggests that incremental system growth tends to remain constant (statistically invariant) or to decline, because developers need to understand the program's source code and behavior. A corollary is often presented, stating that releases that introduce many changes will be followed by smaller releases that correct problems introduced in the prior release, or restructure the software to make it easier to maintain [9].

Prior work by Lawrence [7] used the net module growth as a metric, and found the growth to be statistically random. As mentioned in Section 4.3, the net module growth for our programs (e.g., the one observed in Figure 2) is neither invariant, nor decreasing. A second metric we used was the growth rate (percentage of new functions added to a release). We can see in Figure 3 that the growth rate does not subside. Finally, the third metric we used was the total number of changes to program elements (i.e., changes to functions, global variables and types), to be able to capture finer-grained changes that do not result in an increasing or decreasing number of modules. In Figure 4 we plot the total number of changes against release number for Sendmail. Due to lack of space, we omit showing this kind of graph for other applications, but the trends are similar across all programs. Indeed, we find that releases containing many changes tend to be followed by smaller releases. However, we could not detect any decrease in incremental growth, which is most likely an artifact of the super-linear growth hypothesis for open source software; we discuss this issue in detail in Sections 4.8 and 5.

Therefore, we conclude that the conservation of familiarity law is not confirmed for our test programs.

### 4.6. Continuing Growth

This law stipulates that programs usually grow over time to accommodate pressure for change and satisfy an increasing set of requirements. In previous work, different research teams have used different metrics for measuring system size and growth. Lehman et al. [9, 11], Lawrence [7], and Gall [2] use number of modules to quantify program size and measure growth. Paulson et al. [16], Godfrey and Tu [3], and Izurieta and Bieman [4] use LOC. We used both these metrics, plus the number of definitions.

**Lines of code (LOC)** is a widely used metric for program size; it has the advantage that it accounts for varying modules size, and captures intra-module growth. Figure 5 shows the evolution (in kLOC) of six applications, while Bind is presented in Figure 7; each point in the graph corresponds to an official release. When computing LOC, we excluded comments, empty lines, `#pragmas` containing line number information, etc. and only kept actual code.

We can see that the law of continuing growth is confirmed. While we have found several instances of a new release being slightly smaller (in LOC) than the previous release, they were the result of minor cleanups. The only major drop was in the transition from Bind 9.1.0 to 9.2.0a1; the program shrank considerably, from 254 kLOC to 206 kLOC, because the developers performed a significant cleanup: they completely rewrote two components, the OMAPI protocol handler and the configuration parser.

**Number of modules** shows an ever-increasing trend, with small exceptions, an aspect analyzed in Section 4.3.

**Number of definitions.** This metric characterizes pro-

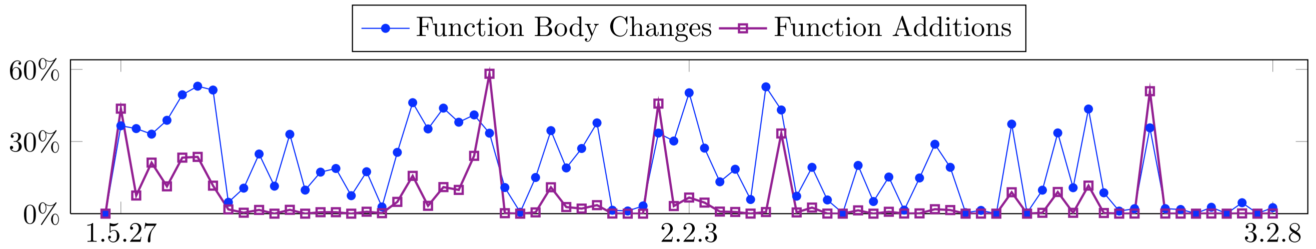


Figure 3: Change and growth rates for Samba.

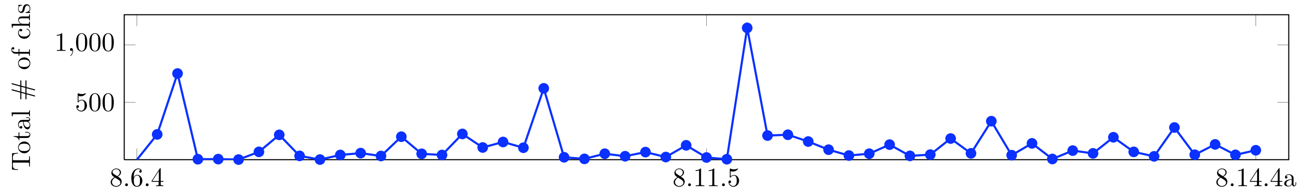


Figure 4: Evolution of total number of changes for Sendmail.

gram evolution in terms of how the total number of definitions (types, global variables, and functions) changes over time. In Figure 1 we can observe system growth since the cumulative number of additions grows faster than the cumulative number of deletions.

In summary, we found that the law of continuing growth is confirmed for our test programs.

### 4.7. Declining Quality

This law stipulates that over time, software quality appears to be declining, unless proactive measures are taken to adapt the software to its operational environment. To understand how software quality changes as software evolves, we use both *internal* and *external* quality metrics.

**External quality** refers to users' perception and acceptance of the software. Since perception and acceptance are difficult to quantify, we rely on defects and vulnerability reports as proxies for external quality. Three of our programs (OpenSSH, Samba, and Quagga) use Bugzilla as their defect tracking system. For each version, we retrieved the Bugzilla data and classified bugs into defects, as described next. To avoid counting spurious defects we only considered those bugs whose status is "verified," "assigned," or "closed," since these have been confirmed by developers. For the bugs whose status is "closed," we only consider those marked as "to be fixed," "fix later," or "won't fix" (i.e., the bug manifestation is due to bugs in other system components).

For Bind, we use MITRE's *Common Vulnerabilities and Exposures* data for all major releases. Unfortunately, for Sendmail, SQLite, and Vsftpd we were not able to find a structured defect database that contains version-specific defect information.

Various metrics have been proposed for measuring the external quality of a release. The first metric is the number of known defects associated with a certain release. For all the programs where defect information was available, we noticed the same trend: major releases tend to have a relatively high number of defects, and the minor releases that succeed them eliminate a certain number of these defects. Over long periods of time, however, we observed that the number of known issues associated with a project tends to decline.

Another quality metric is *defect density*. We computed defect density for each release $i$ using the standard definition, $Defects_i/LOC_i$, and found that the decreasing trend observed in the number of known defects is accelerated when computing defect density because of increasing program size. When using a defect density definition suggested by Mockus et al. [12], $Defects_i/Changes_i$, we found the same decreasing trend.

We take a cue from Paulson et al. [16] and also compute, for each release, the percentage of functions whose bodies have changed. The rationale for this metric is that over time, as defects are found and fixed, less and less functions need to change. For Samba, the evolution of this ratio is illustrated in Figure 3, but we could not spot a clear trend; the same can be said of SQLite. Graphs for rest of the programs show a slightly declining ratio. Note that Paulson et al. [16] have found that this ratio declines for the open source programs they analyzed (Linux, Apache and Gcc).

**Internal quality.** While many metrics have been proposed for assessing internal quality, we limit our study to

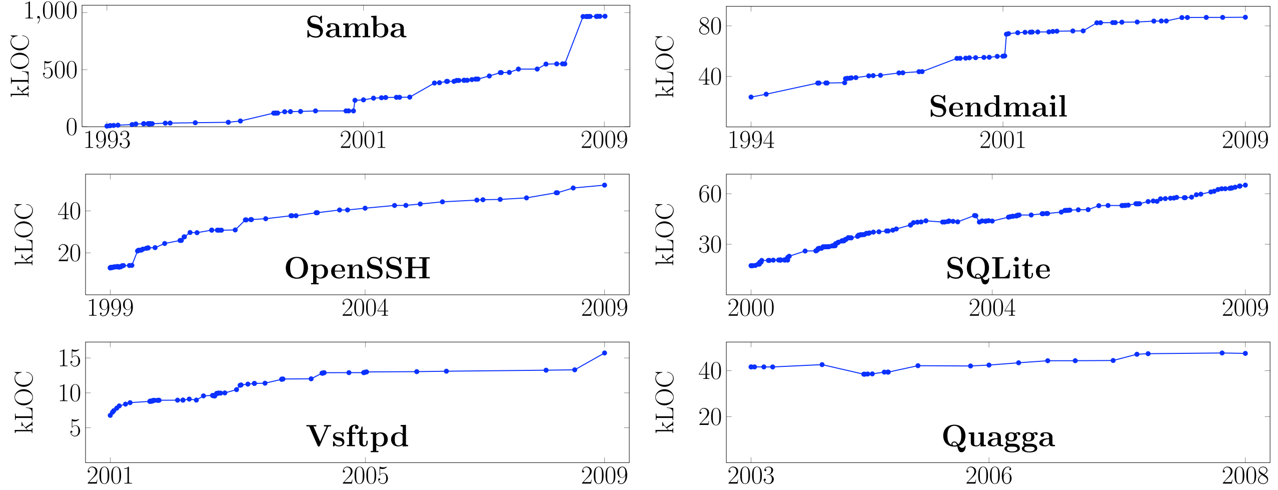


Figure 5: Evolution of application size.

a characterization of software complexity. Since complex software is hard to evolve, we are trying to find out if the software's internal quality is declining by measuring how its complexity changes over time. In Section 4.2 we showed that absolute values for complexity tend to increase, normalized values' decline is due to size increases, and average module size increases as well, which makes the software harder to maintain.

To conclude, when considering both external and internal quality metrics for our test programs, the law of declining quality is not confirmed.

### 4.8. Feedback System

Starting from the law of self-regulation (Section 4.3), Turski [18] came up with a model of system growth similar to feedback in system dynamics. Lehman et al. [11] then formulated the law that software projects are self-regulating systems with feedback. More precisely, this law states that $S_i$, the size of system in modules, can be described in terms of $S_{i-1}$, the size of the previous release, and $E_i$, the effort for that release: $S_i = S_{i-1} + \frac{E_i}{S_{i-1}^2}$.

Later, Turski [19] showed that, assuming the rate of growth is inversely proportional to system complexity, we can obtain a closed-form solution of this equation that expresses the number of modules $S$ as a function of release sequence number (RSN): $S = a\sqrt[3]{RSN} + b$. Prosaically, this feedback dynamic can be expressed as "the system growth slows down over time".

To verify this law, we first perform a linear regression where the independent variable is $\sqrt[3]{RSN}$ and the dependent variable is system size in modules. The results are presented in Table 3.

Second, we compute the growth rate as the derivative of size with time (note that we use time here instead of RSN to account for variance in the intervals between releases). We use $S$ to denote size, so the growth rate is $\frac{\Delta S}{\Delta t}$. If the law was true, then the growth rate $\frac{\Delta S}{\Delta t}$ should be proportional to the first derivative of $a\sqrt[3]{t} + b$, i.e., $\frac{\Delta S}{\Delta t} \approx t^{-2/3}$. We used three metrics for $S$: number of modules, LOC and number of functions. In Figure 6 we plot $\frac{\Delta S}{\Delta t}$ for Samba, with $S$ being number of modules. The graph indicates a largely-varying, positive first derivative and has three spikes that we clip for legibility (values are 1.0, 3.53, 0.54). This suggests a steady growth rate, and certainly violates our expectation that the

| **Program** | **System size (modules)** | |
|---|---|---|
| | $\beta$ | $R^2$ |
| Samba | 176.806 | 0.824 |
| Sendmail | 36.747 | 0.671 |
| Bind | 21.377 | 0.747 |
| OpenSSH | 26.721 | 0.781 |
| SQLite | 9.939 | 0.735 |
| Vsftpd | 4.212 | 0.766 |
| Quagga | 4.193 | 0.712 |

Table 3: Slope and correlation coefficients showing how system size correlates with $\sqrt[3]{RSN}$.

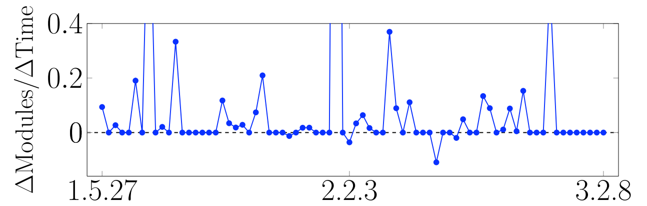


Figure 6: Module growth rate for Samba.

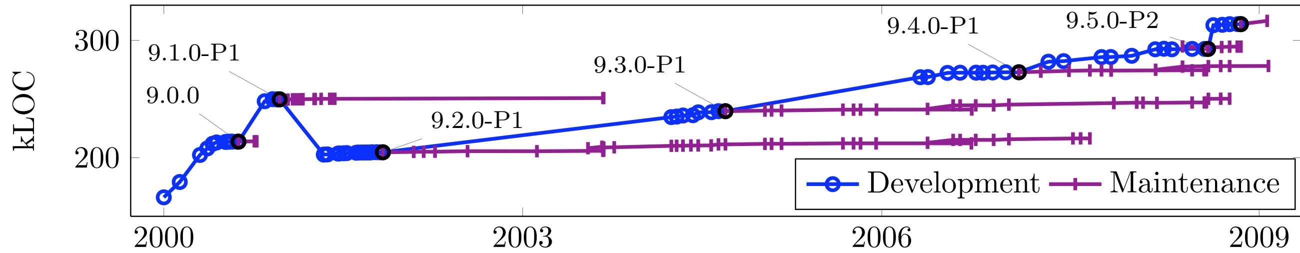


Figure 7: Bind: co-evolution of development and maintenance branches.

graph should have a sub-linear, steady decline, which is the expected behavior of $t^{-2/3}$. While here we use number of modules for system size, the graphs that use LOC and the number of functions look similar.

To conclude, while the system size ripples mentioned in Section 4.3 are consistent with the behavior of dynamic systems with feedback, the growth rate is not, so we could not confirm this law for the applications we examined.

## 5. Observations

We now present our own observations on software evolution, based on analyzing the seven applications outside of the framework of Lehman's laws.

### 5.1 Parallel Evolution

All our applications have points in their history where the development "forks" into a development branch and a stable (maintenance) branch. The development branch forms the "bleeding edge" where new idea and features are introduced and tested. The stable branch will mostly incorporate bug fixes. Periodically, the development branch becomes subject to forking itself. While parallel evolution requires more effort than having a single line of development, maintenance branches are popular with users that prefer stability.

Nakakoji et al. [13] actually show that open source software projects exhibit a variety of development and co-evolution models, from using a single branch (e.g., the GNU family) to parallel branches that co-evolve (e.g., the Linux kernel). Godfrey and Tu [3] found that the size of the Linux kernel, in LOC, grows quadratically with time, if we only consider the development releases. On the other hand, Izurieta and Bieman [4], looking at the evolution of stable branches in FreeBSD and Linux, found the growth (within a branch) to be linear.

Our findings are consistent with these separate growth hypotheses. We illustrate parallel evolution on Bind's development and maintenance branches in Figure 7. The fork points are marked with the release number where the development branch splits. At a fork point, by following the dotted line we find the development branch, whereas to the right of the fork point we have the maintenance branch, i.e., 9.X.0 are development versions, while 9.X.1, 9.X.2, etc. are maintenance versions. We can see that the growth of the development versions (the dotted line) is super-linear, while the growth of maintenance versions (solid lines) is at most linear. We found the same trends in Samba (see Figure 5), but omit the parallel evolution graphs due to lack of space. The other five programs employ parallel development, but to a lesser extent.

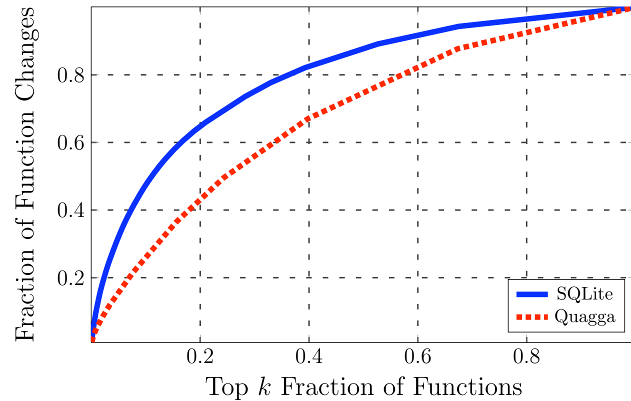


Figure 8: Distribution of changes to functions.

Note that development for a project that exhibits super-linear growth will require an ever-increasing amount of resources and cannot continue ad infinitum. Mockus et al. [12] point out that the usual solution to this is to split the project, or move certain parts into smaller, satellite projects.

### 5.2 Distribution of Changes

One important factor in program evolution is understanding which parts of the code change, and how frequently. Analyzing the reasons that lead to "hot spots," i.e., parts that change frequently, can facilitate evolution. For example, if one such hot spot is due to poor design, we might decide to perform a redesign that facilitates future changes. Moreover, concentrated changes harm parallel development, because developers have to work on the same functions or

modules. In Figure 8 we present the distribution of changes to functions (signature and body) for SQLite and Quagga. The other five programs' distributions lie between these curves: OpenSSH, Samba, and Sendmail display distributions similar to SQLite, while Bind and Vsftpd display distributions similar to Quagga, so we omit them for legibility. SQLite makes very concentrated changes, with 20% of the functions contributing to about two thirds of all changes. On the other hand, in Quagga, two thirds of the changes come from 40% of the functions. Therefore, we believe that SQLite, OpenSSH, Samba, and Sendmail are likely to contain more hot spots.

### 5.3 Interface vs. Implementation

We are also interested in how the ratio of interface changes vs. implementation changes evolves over time, since changes to the interface indicate an actively evolving system. For each version, we computed the ratio $\frac{\text{interface changes}}{\text{interface changes}+\text{implementation changes}}$ using data on changes to function signatures and function bodies, and found this ratio to be small. When also computed the mean ratio across all versions of each application, and found that the mean ranges from 3.3% for Quagga up to 6.42 % for Samba, which suggests that the interface is much more stable than the implementation. Moreover, we found that for all programs except SQLite, this ratio is higher in the initial phases of a program's evolution. This suggests that the architecture of SQLite is still actively evolving, while the other program's architectures have stabilized.

## 6. Threats to Validity

We now discuss possible threats to the validity of our study. *Construct validity* relies on the assumption that our metrics actually capture the intended characteristic, e.g., that LOC, or the number of modules, accurately model system size. We intentionally used multiple metrics for each law to reduce this threat.

We tried to ensure *content validity* by only considering official releases, and analyzing as long a time span in a program's lifetime as possible. We believe that considering individual commits, rather than official releases, would threaten content validity because it exposes "jitter," i.e., experimental features that never make it into official releases, or debugging statements. We acknowledge that for Quagga and Sendmail, our inability to process early versions of the software affects content validity—perhaps in the early stages of development, these programs' evolution trends are different than trends observed later. We also point out the lack of reliable defect information for Sendmail, Vsftpd and SQLite, which affect the validity of our conclusions regarding external quality.

*Internal validity* relies on our ability to attribute any change in system characteristics, e.g., size, to the time lapse between releases, rather than accidentally including or excluding files, modules, etc. We tried to mitigate this threat by (1) making sure we can compile and run each release we are analyzing, and (2) manually inspecting the releases showing large gains (or drops) in the value of a metric, to make sure the change is legitimate.

*External validity* (i.e., the results generalize to other systems) is also threatened in our study. We have only looked at open-source software written in C. Therefore, it is difficult to claim that the results generalize to proprietary software, or software written in other languages.

## 7. Related Work

Gall et al. [2] studied the evolution of a 10 MLOC telecommunication switch software over 20 releases and 21 months. They found that the system size, in number of modules, grows linearly, but modules exhibit vastly different growth rates; in particular one module grows at a much higher rate than others, which is masked when looking at the whole system. This underscores the importance of studying the evolution of individual modules, an aspect we plan to consider in future work.

Paulson et al. [16] compared the evolution of three open source programs (Apache, Linux kernel, and Gcc) with those of three closed-source programs. Although not explicitly mentioned, the evolution time frame for each program seems to be at most five years. They found the growth of each project to be linear when studying major releases only. Our study reaches a different conclusion (super-linear growth rate) albeit for different projects and by analyzing all the releases; this suggests more studies are needed.

In a study similar to ours, Lawrence [7] analyzed the evolution of seven projects over 3–9 years. Their goal was to verify Belady and Lehman's evolution laws [1], i.e., the first five laws in our study. Using metrics such as number of modules, modules changed per release, and number of modification requests, their study found little evidence in support of the laws, except for the first law, continuing growth. They indicate that more precise operational definitions for the laws are needed. We used a variety of metrics in an attempt to improve the precision of these definitions.

Wu and Holt [20] analyzed the evolution of PostgreSQL (85 versions, 7 years) and the Linux kernel (368 versions, 7 years). They use metrics similar to ours (common couplings, calls per function, functions additions/deletions, references to global variables) and found that the two systems clearly observe the laws of continuing growth and continuing change. PostgreSQL shows signs of increasing complexity, while for Linux the results were inconclusive. While one of their systems (the Linux kernel) was larger

than any of the programs we analyzed, we used a larger variety of programs, with longer release histories, which can provide additional insights and a broader perspective. Also, our study tries to verify all Lehman's laws.

Godfrey and Tu [3] examined the evolution of the Linux kernel (6 years, 96 releases). They use LOC as a metric, and conclude, just like us, that Lehman's fourth law (invariant work rate) does not appear to hold for open source software, and that the growth of development releases is super-linear.

Izurieta and Bieman [4] examined 8 years in the lifetime of FreeBSD and 11 years in the lifetime of Linux, but they separate their analysis into stable and development branches. Their conclusion is that growth on individual branches is at most linear, but when considering multiple branches, growth can appear super-linear due to abrupt transitions between the size of a development (or stable) branch and the size of the branched it forked off from. We provide further support for their conclusion.

## 8. Conclusions

In this paper we conduct an empirical study on the evolution of seven long-lived, popular open source programs. The first part of our study investigates Lehman's evolution laws, some of which were formulated by Lehman et al. more than thirty years ago in the context of proprietary software. The results indicate that *Continuing Change*, *Increasing Complexity*, *Self Regulation*, and *Continuing Growth* are still applicable to the evolution of today's open source software. We could not validate *Conservation of Organizational Stability*, *Conservation of Familiarity*, *Declining Quality*, and *Feedback System* for two reasons: (1) lack of process data for the open source projects we examined, and (2) imprecise operational definitions for hypotheses, relying on proxy measurements and yielding inconclusive results.

The second part of our study investigates open source evolution aspects outside the framework of Lehman's laws. We find that different branches of open source programs evolve in parallel, which confirms the parallel evolution hypothesis proposed by other researchers. In addition, all examined programs exhibit "change hot spots," i.e., a high percentage of changes are concentrated to a small percentage of code. Finally, we found that interface changes are much less frequent than implementation changes, and tend to occur towards the initial phases of program evolution.

We believe that our study leads to a better understanding of software evolution, and hence has the potential to advance the state of the practice in software development and maintenance. In future work, we plan to focus on understanding the underlying reasons why some hypotheses hold while others do not, and on proposing solutions for coping with the continuous increases in program size and program complexity that characterize software evolution.

## References


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

[2] H. Gall, M. Jazayeri, R. Klösch, and G. Trausmuth. Software evolution observations based on product release history. In *ICSM*, pages 160–166, 1997.

[3] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *ICSM*, pages 131–142, 2000.

[4] C. Izurieta and J. Bieman. The evolution of FreeBSD and Linux. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 204–211, 2006.

[5] C. F. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Trans. Softw. Eng.*, 25(4):493–509, 1999.

[6] J. Koskinen. Software maintenance costs. `http://users.jyu.fi/~koskinen/smcosts.htm`.

[7] M. J. Lawrence. An examination of evolution dynamics. In *ICSE*, pages 188–196, 1982.

[8] M. Lehman. Laws of Software Evolution Revisited. In *European Workshop on Software Process Technology*, 1996.

[9] M. Lehman, D. Perry, and J. Ramil. On evidence supporting the FEAST hypothesis and the laws of software evolution. In *METRICS '98*, pages 84–88, 1998.

[10] M. Lehman and J. Ramil. Rules and Tools for Software Evolution Planning and Management. *Annals of Software Engineering*, 11(1):15–44, 2001.

[11] 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 *METRICS '97*, pages 20–32, 1997.

[12] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, 2002.

[13] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye. Evolution patterns of open-source software systems and communities. In *IWPSE*, pages 76–85, 2002.

[14] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Mining Software Repositories (MSR)*, pages 1–5, May 2005.

[15] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. *LNCS*, 2304:213–228, 2002.

[16] J. W. Paulson, G. Succi, and A. Eberlein. An empirical study of open-source and closed-source software products. *IEEE Trans. Softw. Eng.*, 30(4):246–256, 2004.

[17] M Squared Technologies - Resource Standard Metrics. `http://msquaredtechnologies.com/`.

[18] W. M. Turski. Reference model for smooth growth of software systems. *IEEE Trans. Softw. Eng.*, 22(8):599–600, 1996.

[19] W. M. Turski. The reference model for smooth growth of software systems revisited. *IEEE Trans. Softw. Eng.*, 28(8):814–815, 2002.

[20] J. Wu and R. Holt. Linker-based program extraction and its uses in studying software evolution. In *Proceedings of International Workshop on Unanticipated Software Evolution*, pages 1–15, 2004.