Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw.: Evol. and Proc. 2013; **25**:193–218
Published online 1 September 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.564

Towards a better understanding of software evolution: an empirical study on open-source software

Iulian Neamtiu*,†, Guowu Xie and Jianbo Chen

Department of Computer Science and Engineering, University of California, CA, USA

SUMMARY

Software evolution is a fact of life. Over the past 30 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 nine open-source projects. Our analysis covers 705 official releases and a combined 108 years of evolution. We first tried to confirm Lehman's eight laws of software evolution on these projects using statistical hypothesis testing. Our findings indicate that only the laws of continuing change and continuing growth are confirmed for all programs, whereas the other six laws are violated by some programs, or can be both confirmed and invalidated, 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. Copyright © 2011 John Wiley & Sons, Ltd.

Received 8 November 2009; Revised 5 June 2011; Accepted 8 July 2011

KEY WORDS: software evolution; Lehman's laws; empirical studies; open source

1. INTRODUCTION

Software continues to evolve long after the first version has been shipped. Numerous estimates indicate that the costs associated with software maintenance and evolution range from 50 to 90 per cent of total costs [21,40,9], whereas others place it at several times the cost of the initial software version [42]. As yearly global software revenues have recently amounted to over $495 billion [41], any factor that can reduce evolution costs is going to have a significant beneficial impact. To reduce software production costs, both managers and developers must understand the factors that drive software evolution and take proactive steps that facilitate changes and ensure that 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 Bison, Bash, BIND 9, OpenSSH, Samba, SQLite, and Vsftpd, as well as the past 15 years of Sendmail and the past 5 years of Quagga. In total, our study covers 705 official releases and over 108 years of cumulative program evolution.

In the first part of our paper, we set out to confirm 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 [3], these laws essentially characterize the software evolution process as a self-regulating and self-stabilizing

*Correspondence to: Iulian Neamtiu, Department of Computer Science and Engineering, University of California, Riverside, CA, USA.
†E-mail: neamtiu@cs.ucr.edu

Copyright © 2011 John Wiley & Sons, Ltd.

system, subject to continuing growth and change [23,24,26]. 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.

We use metrics derived from source code, project and defect information to operationalize each law (e.g., analyze software growth, characterize software changes, and assess software quality), and statistical hypothesis testing to verify whether the law is confirmed or not. For most laws, we used multiple metrics to reduce threats to construct validity. The results of our study indicate that laws I and VI are confirmed, whereas for the remaining six laws, we either found evidence to the contrary, or a more precise operational definition is needed. We present details on our findings in Section 5. To our knowledge, ours is the first study (outside of Lehman *et al*.'s work) 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*. [25], that is, separating the characterizations of system growth and system change.

In the second part of the paper (Section 6), 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, for those applications where the growth rate is super linear on the main development branch, growth is 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, that is, the majority of changes are concentrated to a 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 first puts our work in context by presenting related work (Section 2), then presents an overview of the applications (Section 3) and the methodology—data collection, metrics and hypothesis testing—we followed in our study (Section 4); next, we provide an examination of Lehman's laws (Section 5), offer some of our observations on software evolution outside the framework of Lehman's laws (Section 6), discuss possible threats to validity (Section 7), and present several consequences for researchers and practitioners that emerge from our study (Section 8).

2. RELATED WORK

Fernández-Ramil *et al*. [5] performed a meta-analysis on several empirical studies of how open-source software evolves, and whether Lehman's laws, derived from analyzing proprietary software evolution, apply to open-source evolution. They concluded that three of Lehman's laws ("I - Continuing change", "VI - Continuing growth", and "VIII - Feedback system") apply to open source software evolution, whereas for the other laws, there is evidence to the contrary, or the laws are difficult to verify. Our study also finds that laws I and VI apply to all the programs we examined, and that for some laws, there is evidence to the contrary (or the law is difficult to verify); however, we could not confirm law VIII.

In a study similar to ours, Lawrence [22] analyzed the evolution of nine projects, four operating systems and three batch processing systems, over 3–9 years. Their goal was to verify Belady and Lehman's evolution laws [3], that is, 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. They concluded that, part of the reason why the laws cannot be validated is the lack of precise operational definitions for "complexity" and "changes," which leaves it up to the designer of the study to come up with precise metrics for measuring these. Our study is similar in that we studied large programs over a long time, and we found that some of the laws are not validated. However, in addition to metrics on modules, we also use more fine-grained metrics for measuring change (functions, types, and variables rather than the number of modules), which we believe strengthens the results.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Antoniol *et al*. [1] have studied the evolution of program lexicon (identifiers) and the evolution of structural stability (based on similarity metrics for program entities). Their analysis covers Eclipse (19 versions, 5 years), Mozilla (24 versions, 15 years), and CERN/Alice (13 versions, couple of years); Eclipse is written in Java whereas the other two programs are in C++. They found, just like us, that initial versions are more unstable and subject to changes of a large amplitude, but as software matures, the number and amplitude of changes decrease.

German [8] used a thorough evolution analysis method to recover multi-faceted information on how the Ximian Evolution mail client has changed over time. Their time span covers 1997–2003 (although Ximian Evolution proper started in 2000). Their method looked at source code versions, CVS logs, mailing list, and ChangeLog files. Just like us, they plot the number of files and LOC evolution; their analysis reveals that Ximian Evolution grows at sub-linear rate. They also study how MRs were distributed across time, which MRs affect which modules, which developer contributes to which modules. We studied ChangeLogs, but only to recover defect information, for example, number of bugs associated with each release. Our study is limited to validating Lehman's laws, rather than aiming to be a comprehensive recovery of all the software trails left during evolution.

Gyimóthy *et al*. [13] performed an empirical study on object-oriented software written in C++, aimed to validate several hypotheses that link source code metrics for a class (e.g., number of member functions, depth in the inheritance hierarchy, degree of coupling, degree of cohesion, lines of code) to how fault-prone that class is. Their study was based on source code and bugs in Mozilla (June 2002–June 2004). They found that high coupling, low cohesion, and high lines of code (LOC) for a class are good predictors for high defect density in that class. Our work was focused on C, rather than C++, and we did not aim to construct predictor model, hence we did not correlate coupling, cohesion, or LOC with defects.

Herraiz *et al*. [17] have analyzed the evolution (LOC, number of changes, and number of files) for 3821 libre projects in SourceForge.net; median values for project age (last versus first commit) was 29 months, median LOC was 21,168, and median number of files was 142. Their goal was to test whether the evolution of libre projects is governed by self-organized criticality (SOC) dynamics, that is, whether there are long-range correlations (persisting influences) in the time series of changes made to each project. Their findings suggest that libre projects do not follow the SOC dynamics, but rather the correlations are mostly short-term. The SOC hypothesis is somewhat similar to our test for Law VIII (Feedback System), which we found did not hold. Their methods (time series) are different, they have a breadth-oriented focus and a statistically sounder method (3821 projects); our study has a depth-oriented focus (more metrics for nine long-lived projects).

In another work, Herraiz *et al*. [15] have tried to construct software evolution size predictor models based on time series. They analyzed three large projects: FreeBSD (13 years, 1.4MLOC in the last release), NetBSD (13 years, 2MLOC in the last release) and PostgreSQL (10 years, 290KLOC in the last release). They show that a linear growth model, for example, Size (LOC) $\propto$ days-since-inception is very appropriate for these projects, similar to our findings for several projects (Section 6.1). They show that predicting size using time series works better than predicting size using a linear model.

Kim *et al*. [20] analyzed the evolution of function signatures in seven large, long-lived C projects: Apache 1.3 (9 years), Apache 2.0 (4 years), Apache Portable Runtime (6 years), APR Utility (6 years), CVS (9 years), GCC (5 years), and Subversion (4 years). They introduced a taxonomy of possible signature changes to C functions, showed that complex type name changes are the most frequent kind of signature change, and found evidence that signature changes induce bugs, more so than non-signature changing changes. Just like us, they compute the body-to-signature change ratio and found this to be between 3.5 and 14.9. For our projects, this ratio was higher, between 15.5 and 30.3, most likely because we study projects long after they've matured, a period where signature change becomes less frequent. Similar to us (Section 6.2), they compute the distribution of changes to function signatures (we compute the distribution of changed to function bodies), and found this to be similar to a power law.

Fernández-Ramil *et al*. [4] studied the evolution of 11 large programs written in a variety of programming languages: Blender (6 years), Eclipse (6.9 years), FPC (3.4 years), GCC (19.9 years), GCL (8.8 years), GDB (9.5 years), GIMP (10.9 years), GNUBinUtils (9.4 years), NCBITools (15.4 years), WireShark (10 years), and XEmacs (12.2 years). Their evolution metrics were LOC, number of files, committers-month, and number of distinct contributors. Their goal was to determine

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

whether effort estimation, as computed by the COCOMO model, matches actual effort, as measured by committers-month; it turns out it does not. In their study, they observe, just like us, that source code size tends to increase over time, whether measured by LOC or number of files; they compute the growth rate for number of files and found this to be sub-linear, linear, or super-linear, depending on the project.

Kemerer and Slaughter [19] performed an evolution study by analyzing fine-grained changes (more than 25,000 changes to a system about 3,800 modules over more than twenty years) to financial software written in COBOL. Their study focused more on a taxonomy of changes (e.g., corrective, adaptive, perfective) whereas our study focuses on code metrics; technically, they collect some code-level metrics but do not analyze the evolution of these metrics over time. Also, our analysis does not consider individual commits, but rather official releases, because some code is never actually integrated. Finally, we studied programs written in C because a large base of long-lived open-source software is written in C.

Wu and Holt [46] used a linker-based analysis method to study the evolution of PostgreSQL (85 versions, 7 years) and the Linux kernel (368 versions, 7 years). They employed 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. Whereas PostgreSQL shows signs of increasing complexity, for Linux, the results were inconclusive. Whereas 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.

Izurieta and Bieman [18] 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 as a result of 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.

Godfrey and Tu [10] examined the evolution of the Linux kernel between 1994 and 2000, comprising 96 releases. Their study used LOC as a metric, and focused on the growth of the kernel, as well as the growth rates of individual subsystems. Their conclusion, just like ours, is that Lehman's fourth law (invariant work rate) does not hold of open-source software; in particular, they found Linux's size to grow quadratically with time. Just like us, they used LOC to measure growth, rather than the number of modules (the measure used by Lehman *et al*.) because there is a lot of variation between module sizes, and using LOC captures intra-module growth. Just like us, they use time on the x-axis rather than version number. Our approach differs in that we also looked at system change in terms of program elements, that is, functions, types and variables, which we believe can provide additional insights. Their study can provide better insights about growth of large programs (the Linux kernel in their study grew from 200 kLOC to more than 2000 kLOC) whereas we studied nine smaller programs, the largest of which grew to about 1000 kLOC.

Scacchi [39] first surveys the existing literature on evolution of free/open-source software (FOSS), then examines how each of Lehman's laws fares in relation to FOSS evolution, and finally outlines possible future directions. The conclusions drawn in their work are consistent with our findings: it is harder to validate Lehman's laws on FOSS than on the commercial software on which they were formulated. This difficulty has multiple causes: (i) the variety of change patterns observed in FOSS evolution; (ii) imprecise operational definitions for metrics, for example, complexity, activity rate, or quality; and (iii) fundamental differences in development philosophy and incentives when comparing commercial development subject to market forces versus FOSS development. Our work addresses some of the questions and future work challenges outlined in their paper.

Gall *et al*. [7] studied the evolution of a 10-MLOC telecommunication switch software over 20 releases and 21 months. They found that system size, in a 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*. [35] compared the evolution of three open-source programs (Apache, Linux kernel, and GCC) with those of three closed-source (commercial) programs. Although not explicitly

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

mentioned, the evolution time frame for each program seems to be at most 5 years. They found the growth of each project to be linear when studying major releases only, and using LOC and number of functions as size metrics. Our study reaches a different conclusion (super-linear growth rate) albeit for different projects and by analyzing all the releases; this suggests that more studies are needed. They also found that, for the projects they analyzed, the complexity of the open-source software projects was higher than the complexity of closed-source software.

Grechanik *et al*. [12] have conducted a 32-question empirical investigation on 2080 randomly chosen Java projects from Sourceforge; they pose four software evolution research questions (number of versions, number of fields/methods/classes added or deleted), and 28 research questions on non-evolutionary aspects (e.g., class metrics, use of language constructs, inheritance hierarchies). Their evolution study spans 2427 versions (on average 1.5 versions per application), so it has more breadth and less depth than our study. Their Java-programs findings are similar to our C-programs findings, and indicate that Java applications tend to add fields/methods/classes more than they delete them.

Our own prior work [32] presents the implementation of ASTdiff (an AST differencing tool for C) as well as results of running a small-scale evolution study on several open-source programs: first 5 years in the lifetime of OpenSSH, first 3 years of Vsftpd, and several snapshots of BIND, Apache, and the Linux kernel. The main goal of that work (as with other AST differencing tools [6]) was to collect and classify source code changes. The scope of our current work is a multi-faceted empirical evolution study, hence much broader. We look at much more data (108 years vs 15 years in that study), we analyze many more software aspects (e.g., metrics for growth, complexity), and we draw conclusions based on a more rigorous statistical analysis.

3. APPLICATIONS

We ran our empirical study on nine open-source applications written in C. We used several criteria for selecting our test applications. First, because we are interested in long-term software evolution, the applications had to have a long release history (5+ years, although some of our programs have in excess of 10 years' worth of releases). 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 (e.g., several major releases in the last year we considered).

Table I presents high-level data on application The second column contains the number of official releases for each program, whereas the rest of the columns present information (version, date, size in LOC, and size in number of modules) for the first and last releases.

We aimed at analyzing complete lifespans for each application. For two applications, Sendmail and Quagga, however, their initial versions are old and could not be analyzed (pre-process or compiled) with our tools, because they use antiquated headers, libraries, or they rely on old versions of GCC.

Table I. Application information.

		First release				Last release			
Program	Releases	Version	Date	Size		Version	Date	Size	
				LOC	Modules			LOC	Modules
Bash	19	1.14	06/1994	36,351	65	4.1	12/2009	93,506	136
BIND	168	9.0.0b1	02/2000	169,306	179	9.6.1b1	03/2009	321,689	249
Bison	33	1.00	05/1988	6873	17	2.4.3	08/2010	41,165	80
OpenSSH	78	1.0pre2	10/1999	12,819	34	5.2p1	02/2009	52,284	106
Quagga	29	0.96	08/2003	41,623	45	0.99.11	09/2008	47,511	52
Samba	89	1.5.14	12/1993	5514	2	3.3.1	02/2009	1,045,928	479
Sendmail	57	8.6.4	10/1993	25,912	30	8.14.4a	01/2009	87,842	98
SQLite	172	1.0	08/2000	17,273	14	3.6.11	02/2009	65,108	59
Vsftpd	60	0.0.9	01/2001	6774	23	2.1.0	01/2009	15,711	38

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

We now provide an overview of each application; in each case, we identify the main program in that application's distribution that constitutes the focus of our study (the rationale for focusing on the main program is described in detail in Section 4).

Bash is the popular Unix shell. According to its change log, the first release available on FTP was 1.14.0 (June 1994). We analyzed its entire lifetime. The Bash source distribution includes the `readline` library, but we omitted this library, as it is also distributed separately from Bash, so we did not consider it an integral part of the shell.

BIND is the leading DNS server on the Internet; we analyzed the main server program in the distribution, called `named`. According to its official history (https://www.isc.org/software/bind/history), BIND development goes back to the early 1980s, with BIND 8, now deprecated, being the last major line of development. The current line, BIND 9, is a major rewrite. We analyzed all the BIND 9 versions.

Bison is the GNU parser generator. Bison had the longest lifetime across all the programs we analyzed, more than 22 years. We analyzed its entire lifetime, from version 1.00 to 2.4.3.

OpenSSH is the standard open-source suite of the widely used secure shell protocols. The suite contains a server, called `sshd`, and various clients and utilities. In our study, we focused on `sshd`. The first official release we could find was 1.0pre2, dating back to October 1999. Since then, `OpenSSH` has grown more than fourfold, from 12,819 LOC to 52,284 LOC over 78 official releases.

Quagga is a tool suite for building software routers that support the RIP, OSPF, and BGP protocols on top of IPv4. Quagga started as a fork of the existing Zebra routing software. Whereas the suite contains several executables (protocol daemons), we focused on the main server, called `quagga`. Similar to Sendmail, we had to stop our analysis at version 0.96 (Aug. 2003) because of configuration and pre-processing problems with earlier versions.

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 1 Dec. 1993. The first official release we could find was 1.5.14, dated 8 Dec. 1993. We analyzed 89 official releases of Samba's main program (the SMB server). As shown in Table I, over the past 15 years, the server grew from 5514 LOC to more than 1,000,000 LOC.

Sendmail is the leading email transfer agent today; we analyzed the main server, `sendmail`. Whereas its initial development goes back to the early 1980s, we had to stop at version 8.6.4 (Oct. 1993) because of configuration and pre-processing problems associated with 17-year-old software.

SQLite is a popular library implementation of a self-contained SQL database engine. While intended to be used as library, it also ships with a "shell" that can be used for command–line interaction. Therefore, we analyzed the evolution of the standalone SQL server that consists of merging the shell and the library. 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 source package only contains the daemon (main program) itself. The first beta version, 0.0.9, was released on 28 January 2001.

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

4. METHODOLOGY

4.1. Data collection

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 program in each release, excluding test programs, or, for server programs, various clients that ship with the server. Finally, we "merged" all the source code that goes into building the program into a single `.c` file, using the CIL merger tool [34], however retaining

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

module information. This strategy ensured that we focused on the evolution of one self-contained, standalone program. Note that the LOC numbers in Table I show the source code size for the program we analyzed, rather than LOC for the entire application. 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 and link the program. We separated overlapping versions that occur because of parallel evolution (the development branch versus maintenance branches), and always considered the development versions for our analysis; however, we do an analysis of growth rate on parallel evolving branches in Section 6.1.

4.2. Metric value computation

We ran two source code analysis tools, ASTdiff and RSM, to compute metric values on the 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, for example, changes to types, global variables, function signatures and bodies. Whereas the core algorithm and some case studies are presented in our previous work [32], for this work, we enhanced ASTdiff to support collecting information about code complexity (e.g., common coupling, function calls per function) and modules. Resource Standard Metrics [38]) is a commercial tool that we used for computing cyclomatic complexity.

4.3. Hypothesis testing

We used statistical hypothesis testing to validate our analyses and the conclusions we draw. We performed four kinds of statistical analysis, depending on the nature of each hypothesis:

- *Increase/decrease test*: To test whether a certain metric grows (or decreases) over time, we perform a univariate linear regression where the dependent variable is the metric value for a release. The independent variable is the number of days since the beginning of the project for that release, or the release sequence number, depending on the particular law we are testing. We then test the β (slope) of the regression; the increase (or decrease, respectively) hypothesis is validated if $\beta > 0$ (and $\beta < 0$, respectively) and p-value < 0.05.
- *Non-zero test*: To test whether a certain metric distribution has non-zero values, we perform a one-sample t-test where the specified value is 0; here our null hypothesis is that the distribution has a mean equal to 0; if we are able to reject the null hypothesis (p-value < 0.05), our conclusion is that the distribution has non-zero values.
- *Invariance test*: As some laws stipulate that a certain metric is invariant over time, we test the invariance hypothesis using Levene's test for equality of variance in two samples: one sample contains the actual metric values for all releases, the other sample has the same mean, size, and no variance, that is, all elements are equal to the mean of the first set; we confirm the hypothesis if Levene's test returns that the two sets have equal variance (i.e., values in the first set are statistically invariant) and p-value < 0.05.
- *Non-linear growth test*: To test whether a certain metric has a non-linear growth model, (e.g., that LOC is proportional to the square root of time since first release) we perform a univariate linear regression where the dependent variable is the metric value for a release (e.g., LOC), and the independent variable is the growth model (e.g., square root of time, where time is the number of days since the beginning of the project for that release). The hypothesis is validated if p-value < 0.05.

The data sets we collected for this study, that is, the results of all analyses, metric values, sources for figures, are available online at http://www.cs.ucr.edu/neamtiu/lehman-data.

5. LEHMAN'S LAWS OF SOFTWARE EVOLUTION

As explained in our comparative analysis in Section 2, the main contribution of our study is that we investigate all eight of Lehman's laws, and, to increase construct validity, we use multiple metrics for each law. In Table II, we summarize the findings of our examination; the first two columns contain

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table II. Summary of our statistical hypothesis testing for each Lehman law and each application. "Y" indicates a law is confirmed, "N" indicates the law is not confirmed.

Law	Hypothesis, metric	Bash	BIND	Bison	OpenSSH	Quagga	Samba	Sendmail	SQLite	Vsftpd
I *Continuing change*	H^{1}: cumulative changes	Y	Y	Y	Y	Y	Y	Y	Y	Y
II *Increasing complexity*	H^{2a}: calls per function	N	Y	N	N	N	N	N	N	Y
	H^{2b}: cyc. complexity (absolute)	Y	Y	Y	Y	Y	Y	Y	Y	Y
	H^{2c}: cyc. complexity (normalized)	N	N	Y	Y	N	N	N	Y	N
	H^{2d}: common coupling (absolute)	Y	Y	Y	Y	Y	Y	Y	Y	Y
	H^{2e}: common coupling (normalized)	N	N	N	N	Y	N	N	N	N
III *Self regulation*	H^{3a}: number of modules	Y	Y	Y	Y	Y	Y	Y	Y	N
	H^{3b}: number of functions	Y	Y	Y	Y	N	Y	Y	Y	Y
IV *Conservation of org. stability*	H^{4a}: changes per day	N	N	N	N	N	N	N	N	N
	H^{4b}: change rate	N	N	N	N	N	N	N	N	N
	H^{4c}: growth rate	N	N	N	N	N	N	N	N	N
V *Conservation of familiarity*	H^{5a}: net module growth	N	N	N	N	N	N	N	N	N
	H^{5b}: growth rate (new functions)	N	N	N	N	N	N	N	N	N
	H^{5c}: number of changes	N	N	N	N	N	N	N	N	N
VI *Continuing growth*	H^{6a}: LOC	Y	Y	Y	Y	Y	Y	Y	Y	Y
	H^{6b}: number of modules	Y	Y	Y	Y	Y	Y	Y	Y	Y
	H^{6c}: number of definitions	Y	Y	Y	Y	Y	Y	Y	Y	Y
VII *Declining quality*	H^{7a}: number of defects	N	N	N	N	N	N	N	N	N
	H^{7b}: defect density (by LOC)	N	N	N	N	N	N	N	N	N
	H^{7c}: defect density (by ΔLOC)	N	N	N	N	N	N	N	N	N
	H^{7d}: internal quality, see Law II									
VIII *Feedback system*	H^{8a}: number of modules $\propto \sqrt[3]{RSN}$	Y	Y	Y	Y	Y	Y	Y	Y	Y
	H^{8b}: $\frac{\Delta S}{\Delta t} \propto t^{-2/3}$ (number of modules)	N	N	N	N	Y	N	N	N	Y
	H^{8c}: $\frac{\Delta S}{\Delta t} \propto t^{-2/3}$ (LOC)	N	N	N	N	N	N	N	N	Y
	H^{8d}: $\frac{\Delta S}{\Delta t} \propto t^{-2/3}$ (number of functions)	N	Y	N	N	Y	N	N	N	Y

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

the laws, the third column shows the hypothesis ID and metrics that we used for each law, and the remaining columns show whether, for a specific metric and a specific application, the law is confirmed ("Y") or not ("N"); by "confirmed," we mean the law is validated according to the statistical hypothesis testing procedure described in Section 4. We could only validate laws I and VI on all applications. The remaining laws are not necessarily contradicted; rather, more precise definitions are needed, or the laws do not apply in the context of open-source development, as we shall explain in the remainder of this section.

We now proceed to presenting, for each law, a detailed account of the hypotheses and metrics we used, and our observations on whether the law is validated, invalidated, or a more precise definition is needed.

5.1. *Continuing change*

The first law postulates that a program must continually adapt to its environment, otherwise, it becomes progressively less useful [24]. 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 [3,22,7], system and module size [25,18,10], function modifications, and complexity [35]. As a metric for this law, we use the cumulative number of changes to program elements (i.e., functions, types, and global variables). Therefore, our hypothesis is:

Hypothesis (H^1): cumulative number of changes to program elements in each release is non-zero.

As shown in Table II, we could validate this hypothesis for all applications (using the *non-zero test* described in Section 4). To illustrate program change over time, in Figure 1 we present the cumulative number of changes over the lifetime of Samba (more than 15 years). 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.

Figure 2 shows how Samba 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 [35,7]. To save space, we only present these graphs for Samba; however, the trends are similar for the other programs; the interested reader is referred to our online data repository (Section 4).

We make several observations on how the nine 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. Whereas the rate of change subsides for

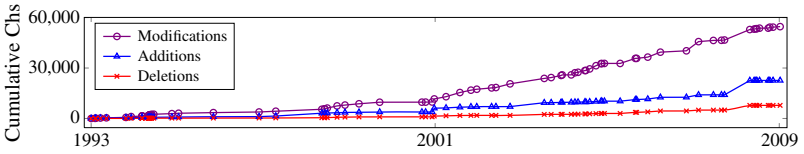


Figure 1. Samba: cumulative changes.

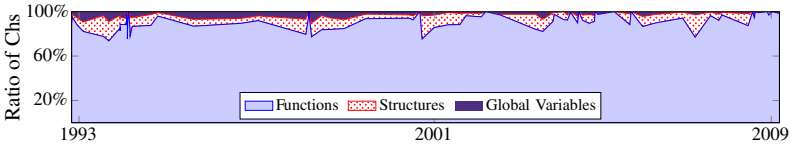


Figure 2. Samba: ratio of changes.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

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 5.6. Third, changes to interfaces are much less frequent than changes to implementation, an aspect we will return to in Section 6.3.

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

5.2. 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 [24].

In an early work by Lehman [3], complexity was defined as the percentage of modules handled relative to the total number of modules; Lawrence [22] uses this definition, as well as programmer productivity. Later work by Kemerer and Slaughter [19] suggests normalized cyclomatic complexity by LOC as a metric, Paulson *et al*. [35] use average function complexity, whereas Wu and Holt [46] employ metrics such as function calls per function and common coupling. To reduce the threat of construct validity, 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 use both absolute and normalized values; the normalized values are computed by dividing the absolute values by the number of possible couplings between modules, that is, $N(N-1)/2$ where N is the number of modules in that version. Therefore, our hypotheses are:

Hypothesis (H^{2a}): average number of calls per function decreases over time.

Hypothesis (H^{2b}): absolute cyclomatic complexity decreases over time.

Hypothesis (H^{2c}): normalized cyclomatic complexity decreases over time.

Hypothesis (H^{2d}): absolute common coupling decreases over time.

Hypothesis (H^{2e}): normalized common coupling decreases over time.

For all hypotheses, $H^{2a} - H^{2e}$, we used the *increase test* described in Section 4, that is, $\beta > 0$. As shown in Table II, the results differ among applications. In addition to Y/N results, in Table III we present the outcome of our linear regression (slope and p-value) where the independent variable is the number of days since the initial release, and the dependent variable is the value of each complexity metric. In several places, p-values above our 0.05 threshold prevent us from validating the law. Regarding function calls per function, for those programs where the p-value is low, for example, BIND, OpenSSH, Sendmail, SQLite and Vsftpd, we observe both negative and positive βs, which

Table III. Slope and p-values showing how program complexity changes over time; p-values less than 0.001 are represented as "0".

	Calls per function (average)		Cyclomatic complexity				Common coupling			
			Absolute		Normalized		Absolute		Normalized	
Program	β	p-val.	β	p-val.	β	p-val.	β	p-val.	β	p-val.
Bash	−2.1e-5	0.567	0.15	0	−3.0e-4	0.001	0.078	0	−6.7e-6	0
BIND	1.7e-4	0	0.39	0	−3.6e-4	0	0.16	0.001	−5.2e-7	0
Bison	−2.2e-4	0.035	0.01	0	5.8e-4	0.002	0.028	0	−1.6e-5	0
OpenSSH	−1.1e-3	0	0.26	0	7.7e-4	0	0.17	0	−1.3e-5	0
Quagga	−2.8e-4	0.003	0.13	0	2.6e-4	0.652	0.044	0	4.4e-6	0
Samba	−9.1e-5	0.137	1.89	0	−1.5e-3	0	0.67	0	−3.6e-5	0
Sendmail	−6.7e-4	0	0.12	0	−3.7e-5	0.367	0.17	0	−3.5e-5	0
SQLite	−1.6e-03	0	0.36	0	3.6e-3	0	0.12	0	−1.6e-5	0
Vsftpd	6.4e-4	0	0.07	0	−4.0e-3	0	0.025	0	−2.9e-5	0

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

suggests both decreasing and increasing trends. Unsurprisingly, we found that the absolute values for cyclomatic complexity and common coupling increase, because program size increases. However, when normalizing common coupling by the number of possible couplings between modules, we get the negative β values in column 10, which suggests that the relative complexity of our test programs (as measured by common coupling) decreases.

Lawrence [22] was the first to point out the necessity of a precise operational definition for testing this law, as, even with commit logs or release notes at hand, it is hard to pinpoint those efforts specifically meant to reduce complexity. In prior work focused entirely to understanding how software complexity changes over time [43], we showed that programmers rarely take steps meant to reduce code complexity; rather, complexity-reducing releases are a by-product of large-scale architectural changes or re-engineering. Moreover, we found that mean module size—an additional complexity metric we used in our related work but not presented here [43]—displays increasing trends as well, for the programs studied.

Therefore, our conclusion is that, as long as complexity appears to increase (which is not the case for all of our test applications), the software structure appears to be deteriorating. The solution would be to provide more precise operational definitions for this law, that is, measures of complexity and identification of complexity-reducing steps.

Because complexity does not appear to always increase over time, our study suggests that this law is not confirmed for all of our test applications.

5.3. Self-regulation

Lehman *et al*. [25] suggest that the evolution of large software systems is a self-regulating process, that is, 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 and function growth for each system. Therefore, our hypotheses are:

Hypothesis (H^{3a}): number of releases with negative adjustments to number of modules is non-zero.

Hypothesis (H^{3b}): number of releases with negative adjustments to number of functions is non-zero.

In Table II, we present the hypothesis testing results (via the *non-zero test*) for each program; in Table V, we present the exact number of shrinking releases (i.e., releases that have negative adjustments). We also present a visual assessment of incremental module growth: Figure 3 shows the module increment on the *y*-axis, whereas the *x*-axis is release number, for each program. We observe that the aforementioned ripples exist indeed, and positive adjustments are more frequent than negative adjustments, for all programs but Vsftpd. The same behavior is observed when considering the number of functions as metric for system size, but in this case, the only program the law was not confirmed on was Quagga. Note that the reason Vsftpd and Quagga do not abide by this law is the absence of negative adjustments in number of modules or number of functions.

Therefore, we conclude that the law of self-regulation is not confirmed for all our test programs.

5.4. 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 lifetime. Lehman *et al*. [25,24] point out the importance of finding accurate metrics for work rate, especially for large projects where communication costs are high. They suggest [25] using the number of changes per release as possible work rate indicator, but leave this to future work. Therefore, we analyze the programs using three definitions for work rate: (i) the average number of changes per day, that is, 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); (ii) change rate [7], that is, the number of function changes divided by the total number of functions; and (iii) growth rate, that is, the number of function additions divided by the total number of functions.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Therefore, our hypotheses are:

Hypothesis (H^{4a}): average number of changes per day is invariant.

Hypothesis (H^{4b}): change rate decreases over time.

Hypothesis (H^{4c}): growth rate decreases over time.

We tested H^{4a} using the *invariance test* described in Section 4; for H^{4b} and H^{4c} we used the *decrease test*, i.e., $\beta < 0$. We found that H^{4a} is not confirmed, i.e., work rate is not invariant. We also found that H^{4b} and H^{4c} are not confirmed, that is, the change and growth rates do not subside (for an example, see Samba's rates in Figure 4), which suggests larger efforts as programs grow. Intuitively, these trends make sense because the programs are open source, and the number of developers tends to increase over a program's lifetime [29]. Note that, in accordance with Lehman's original formulation, we are computing the *per-project work rate*, rather than the *per-developer work rate*—the invariant work rate law, in its original version, was formulated in the context of commercial software development with

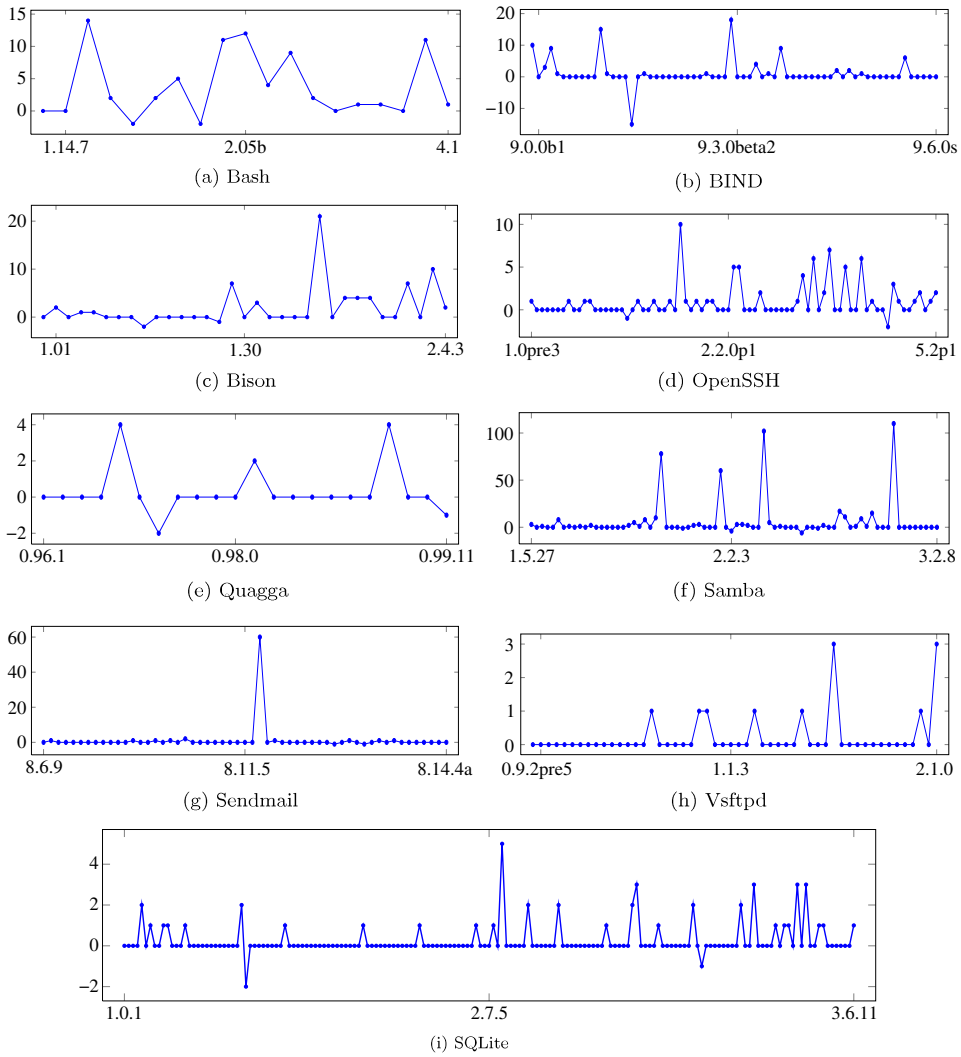


Figure 3. Incremental module growth.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

presumably limited resources and a constant team size. Computing the per-developer work rate would require computation of, and normalization by, developer activity.

Because our domain is open-source software development with increasing team sizes, the fact that this law does not hold is not surprising.

5.5. *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 [25].

Prior work by Lawrence [22] used the net module growth as a metric and found the growth to be statistically random; we used this as a first metric. A second metric we used was the growth rate, expressed as the percentage of new functions added to a release. 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. Therefore, our hypotheses are:

Hypothesis(H^{5a}): net module growth is invariant.

Hypothesis (H^{5b}): function growth rate decreases over time.

Hypothesis (H^{5c}): number of changes decreases over time.

We tested H^{5a} using the *invariance test* described in Section 4; for H^{5b} and H^{5c}, we used the *decrease test*, that is, $\beta < 0$. We found that none of these hypotheses, $H^{5a} - H^{5c}$, are confirmed. As mentioned in Section 5.3, the net module growth for our programs, shown in Figure 3, is neither invariant nor decreasing. The "function additions" graph in Figure 4 illustrates that the growth rate does not subside. In Figure 5, we plot the total number of changes against release number for Sendmail; 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 absolute growth. For some programs, this is a by-product of super-linear growth, as we will discuss in detail in Sections 5.6 and 6.1.

To conclude, the conservation of familiarity law is not confirmed for all our test programs.

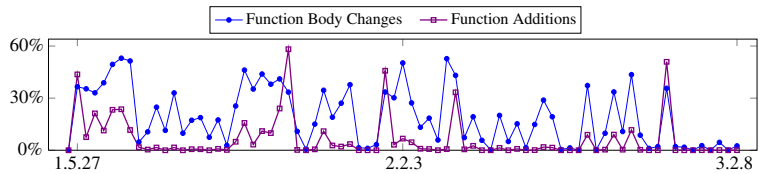


Figure 4. Change and growth rates for Samba.

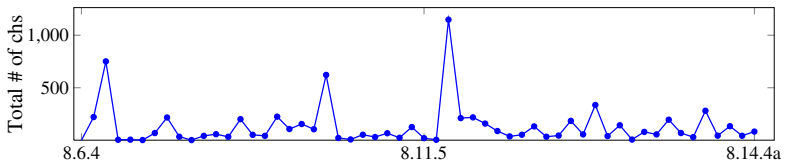


Figure 5. Evolution of total number of changes for Sendmail.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

5.6. *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*. [25,23], Lawrence [22], and Gall [7] have used a number of modules to quantify program size and measure growth. Paulson *et al*. [35], Godfrey and Tu [10], Fernández-Ramil *et al*. [4], and Izurieta and Bieman [18] have used LOC. We use both these metrics, plus the number of definitions.

Lines of code is a widely used metric for program size; it has the advantage that it accounts for varying module sizes and captures intra-module growth. Figure 6 shows the evolution (in kLOC) of our applications; each point in the graph corresponds to an official release. When computing LOC, we excluded comments, empty lines, `#pragmas` containing line number information, and so on, and only kept actual code.

To determine a size evolution model for each application, we used fitting to construct growth trends using the following formula:

$$Size(x) = \alpha x^n + \beta$$

where x is the number of days since the project started and *Size(x)* is the application size, in LOC. In Table IV, we present both the n and the goodness of fit, R^2; we omit αs and βs for brevity. For example, Bash's growth is best approximated by the equation:

$$Size(x) = \alpha x^{0.368} + \beta$$

For BIND and Samba, because of parallel evolution, we show the trends for the development branch, "dev." (our focus), and branches, such as 9.1.X and 2.2.X; the reader can ignore the branch data for now; we will come back to it in Section 6.1. We can see from the table that all programs except Bison and Samba have *sub-linear* growth models (although we explain in Section 6.1 why we believe BIND's evolution consists of two segments, one of which is super-linear).

Number of modules was a metric originally used by Lehman *et al*. when formulating the continuing growth law; hence, we also analyzed the growth of each program in terms of number of modules. We

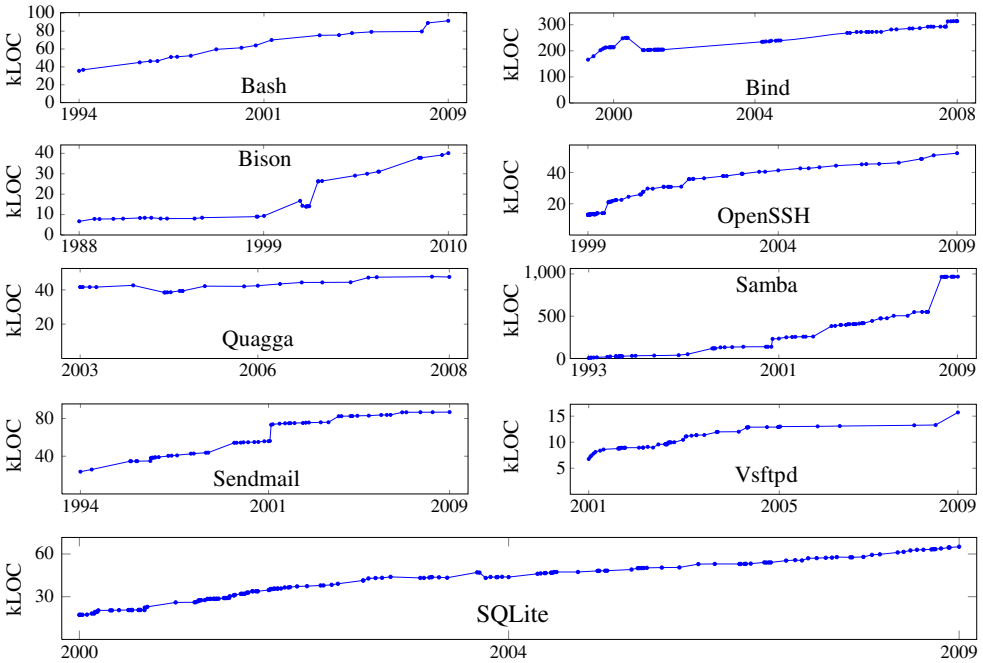


Figure 6. Evolution of application size.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table IV. Growth model coefficients.

Program	n	R^2
Bash	0.368	0.908
BIND		
dev.	0.251	0.828
9.1.X	0.214	0.755
9.2.X	0.127	0.754
9.3.X	0.184	0.831
Bison	1.649	0.938
OpenSSH	0.432	0.987
Quagga	0.063	0.190
Samba		
dev.	2.335	0.978
2.2.X	0.794	0.796
3.0.X	1.036	0.978
Sendmail	0.559	0.960
SQLite	0.348	0.977
Vsftpd	0.367	0.948

have already covered module growth in Section 5.3; Figure 3 presents the incremental module growth for each release. The number of modules shows a generally increasing trend, with some exceptions, as detailed next.

Number of definitions. This metric characterizes program evolution in terms of how the total number of program elements (types, global variables, and functions) changes over time. For example, in Figure 1, we can observe system growth because the cumulative number of additions grows faster than the cumulative number of deletions.

Therefore, our hypotheses are:

Hypothesis (H^{6a}): LOC increases over time.

Hypothesis (H^{6b}): number of modules increases over time.

Hypothesis (H^{6c}): number of definitions increases over time.

For each hypothesis, we used the *increase test* described in Section 4 and found that the hypotheses are confirmed for all programs (Table II).

We also computed, for each program, the number of releases (called "shrinking") that violate this law, that is, the number of releases that have a smaller LOC/number of modules/number of definitions than their immediate preceding release. Table V presents our findings. We can see that the only programs abiding by this law (number of shrinking releases equals 0) are Bash for LOC, Vsftpd for number of modules, and Quagga for number of definitions.

To understand why the law of continuing growth is violated in several releases, we have manually analyzed (source code, change logs) some of these shrinking releases. We have found that, in several instances, a new release is slightly smaller than the previous release as a result of minor cleanups [43]. 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 completely rewrote two components, the OMAPI protocol handler and the configuration parser. Another example is Quagga: in the transition from version 0.96.5 to 0.97.1, Quagga sheds 3000 LOC as a result of the elimination of debugging statements.

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

5.7. 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.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table V. Number of shrinking (smaller than their predecessor) releases.

	Metric		
Program	LOC	Modules	Definitions
Bash	0	2	1
BIND	12	1	4
Bison	4	2	4
OpenSSH	13	2	6
Quagga	6	2	0
Samba	4	4	4
Sendmail	0	2	5
SQLite	18	2	10
Vsftpd	5	0	6

External quality refers to users' perception and acceptance of the software. To quantify perception and acceptance, we rely on the number of defects as a proxy for external quality—an alternative would be conducting interviews with the users of the applications and measuring how their perception and acceptance have changed over time. Note that using the number of defects as external quality proxy threatens construct validity, because a rise in the number of bugs does not necessarily mean a decline in quality—rather, it could be caused by an increased user base, hence an increased number of testers and bug reporters.

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 statuses are "verified," "assigned," or "closed," because these have been confirmed by developers. For the defects 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 caused by bugs in other system components). SQLite has its own, custom, ticket tracking system; to identify defects, we considered the tickets tagged "Active," "Fixed," "Tested," or "Deferred". Bash, Bison, Vsftpd, Sendmail, and BIND do not have dedicated defect tracking systems; therefore, for these programs, we had to manually inspect their release notes/change logs and count the number of defects for each version.

With the defect information in hand, we used several metrics for measuring the external quality of a release. The first metric, the number of known defects associated with a certain release, is shown in Figure 7 (Sendmail is in Figure 8). The only consistent trend across all applications was that 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. However, when adjusting for program size, as described next, all programs show increasing quality.

Another external quality metric is *defect density*, which we illustrate with Sendmail in Figure 8. We computed defect density for each release i using the standard definition, $Defects_i/LOC_i$, and found that it decreases for all programs. When using a defect density definition suggested by Mockus *et al*. [29] that eliminates bias against new, untested code, ($Defects_i/Changes_i$), we found the same decreasing trend.

As a proxy for defects, Paulson *et al*. [35] use the percentage of functions whose bodies have changed—the rationale being that over time, as defects are found and fixed, less and less functions need to change. We computed this percentage (change rate) for each release, as discussed in Section 5.4. For Samba, the evolution of this ratio is illustrated in Figure 4; graphs for the rest of the programs show a slightly declining ratio. Note that Paulson *et al*. [35] have found that this ratio declines for the open-source programs they analyzed (Linux, Apache, and GCC).

Internal quality Whereas many metrics have been proposed for assessing internal quality, we limit our study to a characterization of software complexity. Because complex software is difficult to change/extend and is prone to errors [28,13], we are trying to find out if the software's internal quality is declining by measuring how its complexity changes over time. In Section 5.2, we showed that absolute values for complexity tend to increase, whereas normalized values decline.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

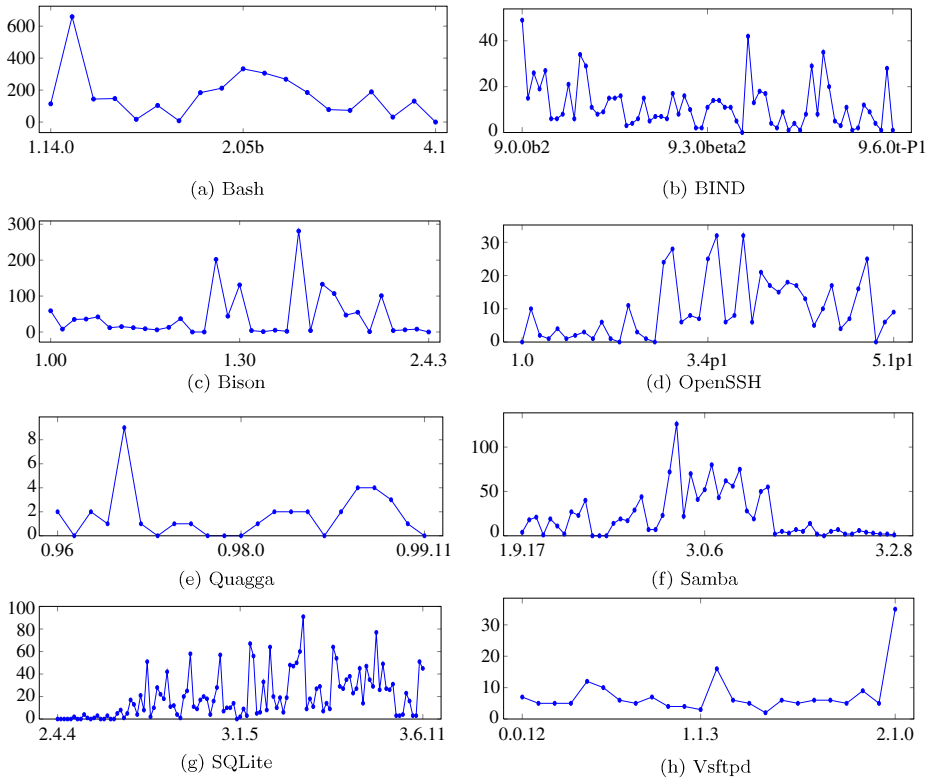


Figure 7. Defects (number of bugs) associated with each release.

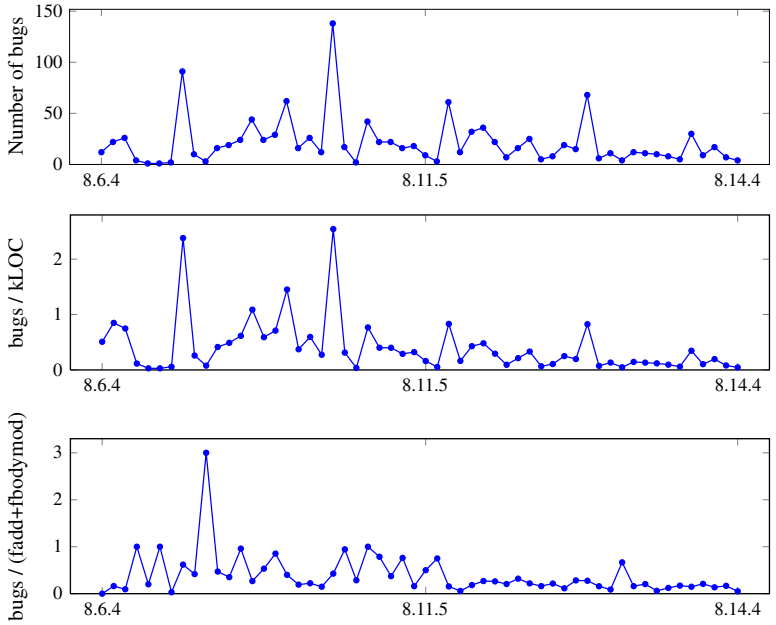


Figure 8. Defects and defect density for Sendmail.

Therefore, our hypotheses are:

Hypothesis (H^{7a}): number of defects increases over time.

Hypothesis (H^{7b}): defect density (by LOC) increases over time.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Hypothesis (H^{7c}): defect density (by ΔLOC) increases over time.

Hypothesis (H^{7d}): internal quality decreases over time.

For hypotheses $H^{7a} - H^{7c}$, we used the *increase test* described in Section 4, that is, $\beta > 0$. As shown in Table II, these hypotheses were invalidated by all applications. For hypothesis H^{7d}, we reuse the results from Law II (Section 5.2), which also indicate the law is not confirmed.

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

5.8. Feedback system

Starting from the law of self-regulation (Section 5.3), Turski [44] came up with a model of system growth similar to feedback in system dynamics. Lehman *et al*. [23] 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 [45] 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: $S = a\sqrt[3]{RSN}+\mathrm{b}$. Put simply, this feedback dynamic can be expressed as "the system growth slows down over time".

Therefore, our hypotheses are:

Hypothesis (H^{8a}): number of modules $\propto \sqrt[3]{RSN}$.

Hypothesis (H^{8b}): $\frac{\Delta S}{\Delta t} \propto t^{-2/3}$, where S = number of modules.

Hypothesis (H^{8c}): $\frac{\Delta S}{\Delta t} \propto t^{-2/3}$, where S = LOC.

Hypothesis (H^{8d}): $\frac{\Delta S}{\Delta t} \propto t^{-2/3}$, where S = number of functions.

For all hypotheses, $H^{8a} - H^{8d}$, we used the *non-linear growth test* described in Section 4. For example, for H^{8a} in our linear regression, the independent variable is $\sqrt[3]{RSN}$, and the dependent variable is system size in modules.

For $H^{8b} - H^{8d}$, we compute the growth rate as the derivative of size with time (note that we use time here instead of release sequence number 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 $H^{8c} - H^{8d}$ were validated, then the growth rate $\frac{\Delta S}{\Delta t}$ should be proportional to the first derivative of $a\sqrt[3]{t}+\mathrm{b}$, that is, $\frac{\Delta S}{\Delta t} \propto t^{-2/3}$ (the results of running a linear regression between $\frac{\Delta S}{\Delta t}$ and $t^{-2/3}$ would show they are related). We used three metrics for S: number of modules, LOC, and number of functions. We found, using the *non-linear growth test*, that H^{8a} is validated on all programs, whereas $H^{8b} - H^{8d}$ could not be validated for all programs. To illustrate the goodness of fit for H^{8a}, in Table VI, we present the slope and R^2 when correlating system size in modules, with $\sqrt[3]{RSN}$ (the p-values, not shown, are all less than 0.001; note that univariate regression analysis is the same as correlation analysis).

To provide a visual assessment of this law, in Figure 9, we plot $\frac{\Delta S}{\Delta t}$ for all applications, with S being the number of modules. The only curve that is somewhat similar to $t^{-2/3}$ is BIND, in Figure 9(c). The rest of the graphs indicate a largely varying, mostly positive first derivative; this suggests a steady growth rate and certainly violates our expectation that the graph should have a sub-linear, steady decline, which is the expected behavior of $t^{-2/3}$. Whereas here we use number of modules for system size, the graphs that use LOC and the number of functions look similar.

To conclude, whereas the system size ripples mentioned in Section 5.3 are consistent with the behavior of dynamic systems with feedback, the growth rate is not; so, we could not confirm this law for all the applications we examined.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

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

	System size (modules)	
Program	β	R^2
Bash	47.065	0.955
BIND	21.377	0.747
Bison	7.358	0.903
OpenSSH	26.721	0.781
Quagga	4.193	0.712
Samba	176.806	0.824
Sendmail	36.747	0.671
SQLite	9.939	0.735
Vsftpd	4.212	0.766

5.9. *Lehman's laws: conclusions*

In Table II, we have presented the findings of our examinations of Lehman's laws for each metric and each application. Note that excepting laws I and VI, for all other laws, we could find evidence to the contrary. However, we refrain from making a sweeping generalization, that is, that the other six laws are invalid, in general, as our study has threats to validity (Section 7). To lend this study more statistical rigor, we would need more projects across a more diverse range, for example, as in Herraiz's work on

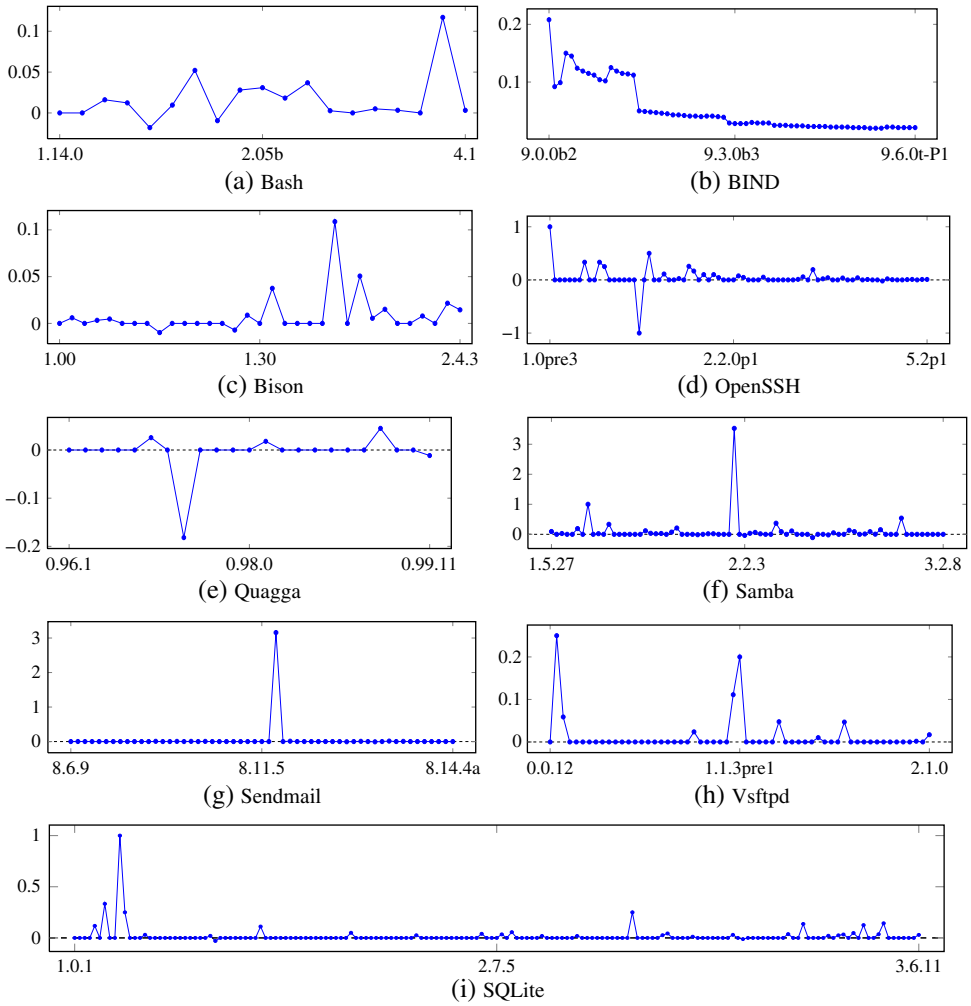


Figure 9. Module growth rate (ΔModules/ΔTime).

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

3821 projects from Sourceforge [17,16], or 13,116 FreeBSD ports [14]; or Grechanik *et al* [12]'s empirical investigation on 2080 randomly chosen Java projects from Sourceforge.

Nevertheless, we believe that presenting a fine-grained account of validating each metric on each application will be valuable to researchers. In addition, as pointed out by Fernández-Ramil *et al*. [5] in their meta-analysis on the applicability of Lehman's laws to open source software evolution, multiple reasons (e.g., the different nature—less structured and more ad hoc—of open source development compared with proprietary development, informal law formulation, discontinuities), prevent the direct application and confirmation of Lehman's laws on open-source systems.

6. OBSERVATIONS

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

6.1. 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 ideas and features are introduced and tested. The stable branch will mostly incorporate bug fixes. Periodically, the development branch becomes subject to forking itself. Whereas parallel evolution requires more effort than having a single line of development, maintenance branches are popular with users that prefer stability.

Nakakoji *et al*. [31] 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 [10] found that, when considering the development releases only, the size of the Linux kernel in LOC grows quadratically with time. On the other hand, Izurieta and Bieman [18], looking at the evolution of stable branches in FreeBSD and Linux, found the growth (within a branch) to be linear. Fernández-Ramil *et al*. [4] found the growth of the development branches (in number of files, not LOC) for 10 large programs to be sub-linear, linear, or super-linear, depending on the program.

To reconcile these different growth models, we have tried to verify the following hypothesis: for programs where the development branch grows super-linearly, growth on the maintenance branch is still at most linear. Our findings confirm this hypothesis. Two programs have significant activity on maintenance branches: BIND and Samba. As we have shown in Table IV, Samba grows super-linearly on the development branch ($n = 2.335$). We computed BIND's growth in isolation for its two segments, before and after the large code deletion in version 9.2.0a1. We found that the growth factors for these two segments were $n = 0.8949$ (versions 9.0.0b1–9.1.0 s-P1) and $n = 1.2581$ (versions 9.20a1–9.6.1b1). Note from Table IV that Samba's maintenance branches grow at most linearly: $n = 0.794$ and $n = 1.036$, respectively, whereas BIND's maintenance branches have ns in the range 0.127–0.214, which supports our hypothesis.

We illustrate this parallel evolution on BIND's development and maintenance branches in Figure 10 and Samba's in Figure 11. The fork points are marked with the release number where the development branch splits. At a fork point, by following the circled line, we find the development branch, whereas to the right of the fork point, we have the maintenance branch, that is, 9.X.0 are development versions,

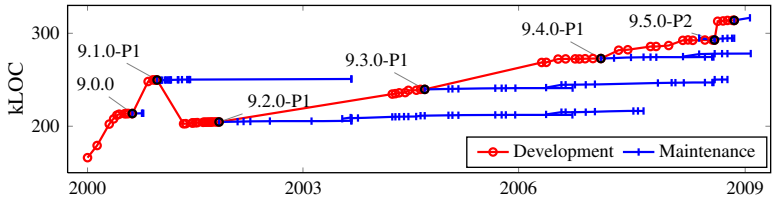


Figure 10. BIND: parallel evolution of development and maintenance branches.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

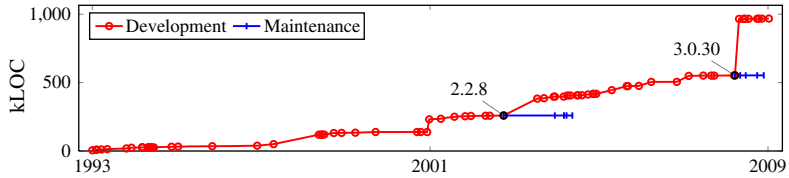


Figure 11. Samba: parallel evolution of development and maintenance branches.

whereas 9.X.1, 9.X.2, and so on are maintenance versions. We can see that the growth of the development versions (circles) tends to be super-linear, whereas the growth of maintenance versions (ticks) is at most linear. The other five programs employ parallel evolution, but to a lesser extent.

On the surface, the development for a project that exhibits super-linear growth will require an ever-increasing amount of resources and cannot continue *ad infinitum*, but open source projects seem to be able to cope with this quite well: Fernández-Ramil *et al*. [4] point out that open-source teams seem to be effective at managing system complexity and keep the project growing super-linearly even after 1 million LOC. Mockus *et al*. [29] point out that the usual solution to this high rate of growth is to split the project, or move certain parts into smaller, satellite projects.

6.2. *Distribution of changes*

One important factor in program evolution is understanding which parts analyzing the reasons that lead to "hot spots," that is, parts that change frequently, can facilitate evolution. For example, if one such hot spot is caused by poor design, the developers might decide to perform a redesign that facilitates future changes. Moreover, concentrated changes harm parallel development, because developers have to work concurrently on the same functions or modules. Finally, code that changes a lot has been shown to be error prone [30,2,11,27].

In Figure 12, we present the distribution of changes to functions (signature and body) for all programs. As we can see, SQLite and Quagga are the extremes. SQLite makes every two thirds of all changes. On the other hand, in Quagga, two thirds of those programs that reveal a more unequal change distribution (SQLite, OpenSSH, Samba, and Sendmail) are likely to contain more hot spots.

6.3. *Interface versus implementation*

We are also interested in how the ratio of interface changes to implementation changes evolves over time, because changes to the interface indicate an actively evolving system. For each version, we computed the ratio $\frac{interface\ \ changes}{interface\ \ changes + implementation\ \ changes}$ using data on changes to function signatures and function bodies, and found this ratio to be small. We also computed the mean ratio across all versions of each application, and found that the mean 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

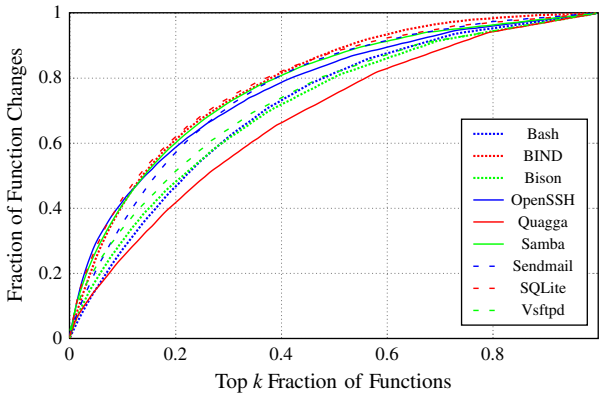


Figure 12. Distribution of changes to functions.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

initial phases of a program's evolution and decreases later on. This suggests that the architecture of SQLite is still actively evolving, whereas the other program's architectures have stabilized.

7. THREATS TO VALIDITY

We now discuss possible threats to the validity of our study. The conclusions we draw from our empirical study are subject to several threats: construct validity, content validity, internal validity, and external validity [36,18].

Construct validity (i.e., independent and dependent variables accurately model the hypotheses) relies on the assumption that our metrics actually capture the intended characteristic, for example, that LOC, the number of program elements, 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," that is, 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.

Internal validity (i.e., changes in dependent variables can be safely attributed to changes in the independent variables) relies on our ability to attribute any change in system characteristics, for example, size, to the time lapse between releases, rather than accidentally including or excluding files, modules, and so on. We tried to mitigate this threat by: (i) making sure we can compile and run each release we are analyzing; and (ii) 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. We have considered servers, a database library, a shell, and a parser generator to broaden the range of application domains for our studied programs; prior work has pointed out that evolution trends might differ significantly across different types of software [47]. However, it is difficult to claim that the results generalize to proprietary software, or software written in other languages.

8. CONSEQUENCES

The main purpose of our study was to examine and report program evolution over long periods of time, rather than provide recommendations for researchers or developers. Nevertheless, we believe it is useful to point out several consequences that emerge from our study. In particular, we focus on two questions: (i) what can researchers do to help construct a better theory of software evolution? and (ii) how can practitioners benefit from the findings of our study and other studies like this?

8.1. Researchers

Laws violations as case studies. In Table II, we have presented the findings of our study for each law, application, and metric. We believe that investigating the reasons why certain laws are violated (the "N" entries) will likely constitute a fruitful research effort. In particular, our study could help researchers choose certain programs as interesting case studies, such as: What led to, and what were the consequences of BIND's large source code drop in version 9.2.0a1? How does Samba manage to sustain a quadratic growth? Why is Vsftpd never deleting modules, or Quagga never deleting definitions?

Closed-source software. The first among Lehman *et al*.'s laws were formulated in the early 1970s, based on data from development of OS/360 at IBM; follow-up studies, for example, by Lawrence [22] looked at commercial software from IBM and other vendors. Replicating our study on closed-source (commercial) software would expand the analysis over more (and very different) development processes, hence increase validity. Paulson *et al*. [35] compared several closed-source and open-source

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

programs and found that closed-source projects exhibited higher internal quality (less complex) but lower external quality (more defects) than open-source ones. Moreover, adherence to process standards within a commercial software development organization might result in a wealth of process data (e.g., effort for each release) that are not available in open-source development.

Other programming languages. One of the threats to the validity of our study is that we only looked at programs written in C. To get a better perspective, we should also look at long-term evolution for programs written in other languages, and compare those observations with the ones presented here. For example, the interface/implementation boundary for C programs is not clear, as opposed to languages where modularity is strictly enforced, such as Java or ML.

Fine-grained change detection. Our study confirms the law of *continuing change*. However, we have limited the granularity of change detection to analyzing how many functions, types, and global variables have changed. We have not measured, or tried to characterize in detail, how types (e.g., `structs` or `typedefs`) change, or how functions change (which kinds of statements, e.g., `if` or `switch` are mostly frequently added, deleted, or changed). In previous work [32], we performed such a fine-grained study, but that study was limited to detecting fine-grained changes to types, and analyzed three programs only. Detecting fine-grained changes is potentially beneficial to a wide array of research areas: bug mining [37], dynamic software updating [33], or constructing IDEs that facilitate software evolution.

8.2. Practitioners

In addition to opening new research avenues, we believe that our study can help and project managers to produce better software that is easier to evolve.

Software developers. The complexity increases and changed hot-spots revealed in our study present evidence that developers should take proactive action to prevent software decay and avoid producing software that is difficult, if not impossible, to repair and evolve. Developers can prevent the trend of ever-increasing code complexity (which, as pointed out in Section 5.2, unfortunately, is the rule, rather than the exception, for the programs we analyzed). By continuously monitoring code complexity and taking proactive steps (e.g., restructuring or refactoring), maintenance costs can be reduced. Similarly, code hot spots (functions or modules that make up the majority of changed entities) lead to poor parallelization and hamper team efforts; and numerous studies show hot-spot code to be error prone [30,2,11,27].

Project managers. This study also helps managers plan their projects more judiciously; as pointed out in Sections 5.6, 5.1, and 5.2, software tends to grow a lot, change a lot, and become more complex. By provisioning resources to accommodate growth and by taking aggressive steps to avoid software decay and prevent complexity build-up, managers can stay on time and on budget. Moreover, as the parallel evolution curves in Section 6.1 show, managers (for commercial or open-source projects) should be prepared to "split" their development in multiple software lines that evolve in parallel. In fact, a study on the evolution of Apache and Mozilla shows that splitting a large project into loosely connected modules is essential for taming complexity and keeping communication costs in check [29].

Finally, we underscore the importance of managers and developers continuously monitoring software quality (e.g., using tools that measure complexity or distribution of changes) to keep the software.

9. CONCLUSIONS

In this paper, we conduct an empirical study on the evolution of nine 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 30 years ago in the context of proprietary software. The results indicate that *Continuing change* and *Continuing growth* are still applicable to the evolution of today's open-source software. We could not validate *Increasing complexity*, *Self-regulation*, and *Conservation of organizational stability*, *Conservation of familiarity*, *Declining quality*, and *Feedback system* for two reasons: (i) lack of process data for the open-source projects we examined; and (ii) imprecise operational definitions for hypotheses, relying on proxy measurements and yielding inconclusive results or results that invalidate the hypotheses.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

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," that is, 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 research and practice in software development and maintenance. In future work, we plan to focus on understanding the underlying reasons why some hypotheses hold whereas others do not, and on proposing solutions for coping with the continuous increases in program size and program complexity that characterize software evolution.

ACKNOWLEDGMENTS

We thank the anonymous reviewers, as well as ICSM 2009 participants, for their helpful comments on the drafts of this paper.

REFERENCE

1. Antoniol G, Gueheneuc Y-G, Merlo E, Tonella P. Mining the lexicon used by programmers during software evolution. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, 2007; 14–23.
2. Arisholm E, Briand LC. Predicting fault-prone components in a Java legacy system. In *ISESE '06: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ACM: New York, NY, USA, 2006; 8–17.
3. Belady LA, Lehman MM. A model of large program development. *IBM Systems Journal* 1976; **15**(3):225–252.
4. Ramil JF, Cortazar DI, Mens T. What does it take to develop a million lines of open source code? In *Open Source Ecosystems: Diverse Communities Interacting, IFIP Advances in Information and Communication Technology*, Boldyreff C, Crowston K, Lundell B, Wasserman A (eds.). Springer: Boston, 2009; **299**:170–184.
5. Ramil JF, Lozano A, Wermelinger M, Capiluppi A. Empirical studies of open source evolution. In *Software Evolution*, 2008; 263–288.
6. Fluri B, Wursch M, Pinzger M, Gall HC. Change distilling: tree differencing for fine-grained source code change extraction. *Software Engineering, IEEE Transactions on* 2007; **33**(11):725–743.
7. Gall H, Jazayeri M, Klösch R, Trausmuth G. Software evolution observations based on product release history. In *ICSM* 1997; 160–166.
8. German DM. Using software trails to reconstruct the evolution of software: research articles. *J. Softw. Maint. Evol* November 2004; **16**:367–384.
9. Ghezzi C, Jazayeri M, Mandrioli D. *Fundamentals of Software Engineering*. Prentice Hall PTR: Upper Saddle River, NJ, USA, 2002.
10. Godfrey MW, Tu Q. Evolution in open source software: a case study. In *ICSM*, 2000; 131–142.
11. Graves TL, Karr AF, Marron JS, Siy H. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering* 2000; **26**(7):653–661.
12. Grechanik M, McMillan C, DeFerrari L, Comi M, Crespi S, Poshyvanyk D, Fu C, Xie Q, Ghezzi C. An empirical investigation into a large-scale Java open source code repository. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, ACM: New York, NY, USA, 2010; 11:1–11:10.
13. Gyimothy T, Rudolf F, Istvan S. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering* October 2005; **31**:897–910.
14. Herraiz I, Gonzalez-Barahona JM, Robles G. Towards a theoretical model for software growth. In *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on*, May 2007; 21.
15. Herraiz I, Gonzalez-Barahona JM, Robles G, German DM. On the prediction of the evolution of libre software projects. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, 2007; 405–414.
16. Herraiz I. A statistical examination of the evolution and properties of libre software. Ph.D. Thesis, Universidad Rey Juan Carlos, 2008. http://purl.org/net/who/iht/phd.
17. Herraiz I, Gonzalez-Barahona JM, Robles G. Determinism and evolution. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, ACM: New York, NY, USA, 2008; 1–10.
18. Izurieta C, Bieman J. The evolution of FreeBSD and Linux. In *ISESE '06: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, 2006; 204–211.
19. Kemerer CF, Slaughter S. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering* 1999; **25**(4):493–509.
20. Sunghun K, Whitehead EJ, Bevan J, Jr. Properties of signature change patterns. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, IEEE Computer Society: Washington, DC, USA, 2006; 4–13.
21. Koskinen J. Software maintenance costs. http://users.jyu.fi/ ~koskinen/smcosts.htm [2 August 2011].

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

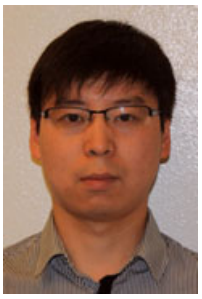
22. Lawrence MJ. An examination of evolution dynamics. In *ICSE*, 1982; 188–196.
23. Lehman MM, Ramil JF, Wernick PD, Perry DE, Turski WM. Metrics and laws of software evolution – the nineties view. In *METRICS '97*, 1997; 20–32.
24. Lehman MM. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, 1996.
25. Lehman MM, Perry DE, Ramil JF. On evidence supporting the FEAST hypothesis and the laws of software evolution. In *METRICS '98*, 1998; 84–88.
26. Lehman MM, Ramil JF. Rules and tools for software evolution planning and management. *Annals of Software Engineering* 2001; **11**(1):15–44.
27. Leszak M, Perry DE, Stoll D. Classification and evaluation of defects in a project retrospective. *Journal of Systems and Software* 2002; **61**(3):173–187.
28. McCabe TJ. A complexity measure. In *ICSE*, 1976; 407.
29. Mockus A, Fielding RT, Herbsleb JD. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology* 2002; **11**(3):309–346.
30. Nagappan N, Ball T. Use of relative code churn measures to predict system defect density. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, ACM: New York, NY, USA, 2005, 284–292.
31. Nakakoji K, Yamamoto Y, Nishinaka Y, Kishida K, Ye Y. Evolution patterns of open-source software systems and communities. In *IWPSE*, 2002; 76–85.
32. Neamtiu I, Foster JS, Hicks M. Understanding source code evolution using abstract syntax tree matching. In *Mining Software Repositories (MSR)*, May 2005; 1–5.
33. Neamtiu I, Hicks M, Stoyle G, Oriol M. Practical dynamic software updating for C. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM: New York, NY, USA, 2006, 72–83.
34. Necula GC, McPeak S, Rahul SP, Weimer W. CIL: intermediate language and tools for analysis and transformation of C programs. *LNCS* 2002; **2304**:213–228.
35. Paulson JW, Succi G, Eberlein A. An empirical study of open-source and closed-source software products. *IEEE Transactions on Software Engineering* 2004; **30**(4):246–256.
36. Perry DE, Porter AA, Votta LG. Empirical studies of software engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, ACM: New York, NY, USA, 2000; 345–355.
37. Raghavan S, Rohana R, Leon D, Podgurski A, Augustine V. Dex: a semantic-graph differencing tool for studying changes in large code bases. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, IEEE Computer Society: Washington, DC, USA, 2004; 188–197.
38. M Squared Technologies – Resource Standard Metrics. http://msquaredtechnologies.com/ [2 August 2011].
39. Scacchi W. Understanding open source software evolution: applying, breaking, and rethinking the laws of software evolution. 2003.
40. Seacord RC, Plakosh D, Lewis GA. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley: 2003.
41. Software Magazine. Software 500. November 2010.
42. Sommerville I. *Software Engineering (*7th edn*)* Pearson Addison Wesley: 2004.
43. Suh SD, Neamtiu I. Studying software evolution for taming software complexity. *Software Engineering Conference, Australian* 2010; 0:3–12.
44. Turski WM. Reference model for smooth growth of software systems. *IEEE Transactions on Software Engineering* 1996; **22**(8):599–600.
45. Turski WM. The reference model for smooth growth of software systems revisited. *IEEE Transactions on Software Engineering* 2002; **28**(8):814–815.
46. Wu J, Holt RC. Linker-based program extraction and its uses in studying software evolution. In *Proceedings of International Workshop on Unanticipated Software Evolution* 2004; 1–15.
47. Zimmermann T, Zeller A, Weissgerber P, Diehl S. Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on* 2005; **31**(6):429–445.

AUTHORS' BIOGRAPHIES

Iulian Neamtiu is an assistant professor in the Department of Computer Science and Engineering at University of California, Riverside. He received his PhD in Computer Science from University of Maryland at College Park. His research interests include software engineering and programming languages, in particular, software evolution and dynamic software updating. He is the principal developer of Ginseng, a dynamic software updating implementation for C that provides certain update safety guarantees, and has been used for constructing and applying on-the-fly updates, based on actual releases, to widely used open-source software.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr



Guowu Xie is a PhD student in the Department of Computer Science and Engineering at University of California, Riverside. His research interests include network traffic monitoring and classification, graph mining, and software evolution. Before starting graduate studies at UC Riverside, he received BSEE and MSEE degrees from Tongji University and Shanghai Jiao Tong University in 2005 and 2008, respectively.

Jianbo Chen is a Master's student in the Department of Computer Science and Engineering at University of California, Riverside. His research interests include data mining, database indexing, and software evolution. Before starting his graduate studies, he worked for Bearingpoint Management and Consulting Company as a software engineer for 2 years. His work was in the areas of data migration, business workflow engine, and plug-in development for Eclipse framework.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/smr