DIU Journal of Science and Technology Volume 12, Issue 1, January 2017

2017-01

Empirical Study of Cyclomatic Complexity and Interface Complexity of Evolving Open Source Systems

Olatunji, Michael A.

Daffodil International University

EMPIRICAL STUDY OF CYCLOMATIC COMPLEXITY AND INTERFACE COMPLEXITY OF EVOLVING OPEN SOURCE SYSTEMS

Michael A. Olatunji, Rufus O. Oladele, Amos O. Bajeh

Department of Computer Science
University of Ilorin, Ilorin, Nigeria
P. M. B. 1515, Ilorin, Nigeria.

E-mail: mikeolaus@gmail.com, roladele@yahoo.com, bajehamos@unilorin.edu.ng

Abstract: *This paper aims at investigating the validity of Lehman's Law of Increasing Complexity. Two metrics namely cyclomatic complexity and interface complexity were defined to capture increasing complexity. The goal was to verify if these metrics can be used to validate Lehman's law of increasing complexity. Empirical analysis was performed using historical data collected on four evolving Open Source Systems (OSS). Results show that the considered Lehman's law is partially supported by the collected data and the metrics. In particular, empirical results reveal that: total cyclomatic complexity and total interface complexity are increasing from version to version; average cyclomatic complexity and average interface complexity either declines or increases within a very short range; and function interface complexity hardly decline in evolving OSS. Also, addition of low complex functions reduces cyclomatic complexity in evolving OSS but does little in reducing function interface complexity.*

Keywords: *Cyclomatic Complexity, Lehman's Law, Empirical Validation, Open Source Systems, Software Metrics, Software Evolution.*

1. Introduction

Software evolution is the process by which a software system is being adapted to its environment as time passes in order to maintain relevance owing to change in requirements. Software evolution has been receiving extensive research attention since the postulation of the first three set of Lehman's laws of software evolution in 1974 as a result of Lehman's study on the IBM commercial systems. The objective of the study that led to the formulation of the laws was to empirically investigate and identify invariant properties of software systems as they evolve over time in order to obtain theory for software evolution. In 1996, Lehman's laws of software evolution became eight in number, owing to continuing studies [1]. Lehman [2] classified all systems under Specified, Problem-solving and Evolutionary (SPE) scheme. There are three classes of systems in this scheme. However, the laws of software evolution were said to be referring only to Evolutionary type (E-type) systems. *E*-type systems are components of the real world and reflections of human processes; they solve problems that involve people or real world. As the world and human requirements change, software requirements often change which invariably lead to modifications in evolutionary software systems. The four evolving OSS used as datasets in this study are E-type systems. Among the eight laws of software evolution, as listed below is the law of increasing complexity which is the focus of this empirical study:

The law of continuing change states that E-type systems must be continually adapted else they become progressively less satisfactory.

The law of increasing complexity states that as an E-type system evolves, its complexity which reflects deteriorating structure increases unless work is done to maintain or reduce it. This Law is the focus of the empirical study in this paper.

The law of self-regulation states that E-type system evolution process is self-regulating with distribution of product and process measures close to normal.

Date of submission : 13.02.2017 Date of acceptance : 01.07.2017

The law of conservation of organisational stability states that the average effective global activity rate in an evolving E-type system is invariant over product lifetime (Invariant work rate).

The law of conservation of familiarity states that as an E-type software system evolves, all associated with it like developers, sales personnel and users must maintain mastery of its content and behaviour to achieve satisfactory evolution. However, excessive growth diminishes that mastery. Hence the average incremental growth remains invariants as the software system evolves.

Other laws are the law of continuing growth which states that the functional content of E-type systems must be continually increased to maintain satisfaction over lifetime; the law of declining Quality states that the quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to the operational environment changes; and the law of feedback system states that E-type evolution process constitute multi-level, multi-loop, multi agent feedback system and must be treated as such to achieve significant improvement over reasonable base.

This study investigates the validity of the law of increasing complexity in the context of open source system (OSS) which are different from the seven commercial systems studied by Lehman in term of development approach. OSS are developed in communities of programmers that are geographically distributed. Unlike in commercial software used by Lehman in which the development and the evolution phases occur consecutively, in OOS the development and the evolution phases occur simultaneously. Several studies have been conducted to validate the laws with contradicting results; some refuting some of the laws while others confirming them. The law of increasing complexity, the focus of this study, is one of the laws that have been both refuted and confirmed [3], [4], [5], [6], [7], [8], [9], [10], [11]. Therefore more studies need to be conducted to increase confidence in the laws and to understand the extent of applicability of the laws. This study is a contribution towards this direction.

Software complexity has immense influence on software external quality attributes such as understandability, analysability and maintainability. The measures for software complexity which are considered in this study are McCabe Cyclomatic Complexity (MCC) and Interface Complexity (IC). MCC is one of the most popular measure of software attribute and it is a measure of decision complexity of the functional units in software system. It quantifies the number of linearly independent route in the control flow graph representing the flow of control of a subprogram [12]. IC is count of the number of parameters and return points in a functional unit of a system [13].

The rest of the paper is organized as follows. Section 2 discusses related works on the validation of Lehman's Second law of software evolution. The sample OSS that are used as case studies are presented alongside the metrics and measurement tools in section 3. In section 4, the results are both presented and discussed. Section 5 highlights likely threats to the validity of the results and how they have been mitigated. The paper is finally concluded in section 6.

2. Related Work

Some of the works which address Lehman second law: law of increasing complexity, which is the focus of this paper are presented in this section as follows.

Pirzada [8] studied research, academic, and supported & commercial streams of UNIX systems; he stated in his PhD thesis that the supported & commercial stream of UNIX operating system validate the laws which of course includes the law of increasing complexity. The other two streams evolutions are not compatible with the laws of software evolution.

Similarly Eick et al. [10] showed that there are source code decays unless effort and resources are allocated to prevent and maintain systems throughout the later stages of their deployment.

Roy and Cordy [14] investigated two evolutionary small-scale open source systems (OSS). They found out that the systems obey the Lehman law

of increasing complexity but with minor exceptions. However, they stated that more empirical work is needed to conclude that small E-type systems are always inclined with the law of increasing complexity.

Israeli and Feitelson [6] analyzed 810 versions of Linux kernel and found out that there were evidences of work been done to reduce code complexity by code improvements and addition of many low McCabe Cyclomatic Complexity functions; however, the number of high complex functions keeps growing significantly and so they could neither refute or confirm the law of increasing complexity.

Fern´andez-Ramil et al. [5] studied different effort models for OSS development, comparing them with effort models used traditionally in closed-source software. They concluded that complexity does not slow down the growth of the analyzed OSS projects, contrarily to what is stated by the laws. The authors suggest that open source software is more effective than closed-source, and therefore less effort is required to develop projects of similar sizes.

Vasa [11] studied 40 large OSS using size and complexity metrics. They found support for the law of continuing change, law of self-regulation, law of conservation of familiarity and law of continuing growth but could not confirm law of increasing complexity.

In a study of 705 releases of 9 open source software projects, Neamtiu et al. [7] could not confirm the law of increasing complexity because not all the 9 systems exhibit increasing complexity in the evolution. They also pointed out that programmers rarely take steps meant to reduce code complexity but they affirm that increasing complexity causes deteriorating structure.

Alenezi and Almustafa [9] studied complexity evolution of five open source projects from different domains. They concluded that size of the systems they studied increases linearly in the course of evolution but the systems average cyclomatic complexities and maximum cyclomatic complexities show no significant increment changes. They argue that the increase in source line of code show increasing complexity but stable cyclomatic complexity shows work is being done to maintain the systems, therefore, their findings confirm Lehman law of increasing complexity.

3. Experiment Design

3.1 Sample OSS (Case Studies)

We downloaded the datasets which are used as case studies for this paper from UCR online software repository. Neamtiu et al.[7] collected and placed 9 merged source codes of evolving OSS written in C language in the UCR online software repository. We used a collected dataset placed on internet repository by a group of evolution researchers because quantification of law of software evolution must be based on empirical results, verifiable and repeatable, and made on a large scale, so that conclusions with statistical significance can be achieved [15]. If software evolution is analyzed with data that is not available to third parties, it cannot be verified, repeated and replicated. It would be erroneous and dangerous to build a theory on empirical studies that do not fulfill those requirements.

The datasets used as case studies for this project are well evolved open source applications, namely: Bison systems, Samba systems, SQLite Systems and Vsftpd systems. The above dataset are selected because they have long term software evolution, sizable and actively maintained.

Table 1 presents the sample OSS used for the empirical analysis.

Table 1 Sample OSS

OSS					**Description**
Bison:	v1.0	v1.3	v1.5	v2.0	Bison is a general-purpose parser generator, which can be used to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages.
SLOC	10943	22637	41059	45266	
No. of Functions	128	223	572	662	
Samba	v1.5.30	v2.0.0	V3.0.9	V3.3.1	Samba is a tool suite that facilitates Windows-UNIX/Linux interoperability. It is based on the common client/serverprotocol of Server Message Block (SMB) and Common Internet File System (CIFS)
SLOC	15564	182901	611722	1615698	
No. of Functions	212	2068	5595	13157	
VSFTPD	v0.0.9	v1.1.2	v2.0.0	v2.1.0	Vsftpd stands for "Very Secure File Transfer Protocol Daemon" and is the File Transfer Protocol (FTP) server in major Linux distributions like Ubuntu, Centos, Unix and the likes. FTP is a standard network protocol used for the transfer of computer files between a client and server on a computer network.
SLOC	25251	40586	71007	65249	
No. of Functions	167	300	782	1284	
SQLite	v1.0.0	v2.0.7	v3.1.1	v3.6.11	SQLite is an implementation of a self-contained Structured Query Language (SQL) database engine. It is a library in C programming language. It is not a client-server database management system and it is widely used for client storage in application softwares.
SLOC	25251	40586	71007	65249	
No. of Functions	167	300	782	1284	

Table 1 gives information on some selected versions of the sample OSS. We analysed the entire versions of the sample OSS available on the UCR online repository as at the time of carrying out this study. Bison has 33 versions (v1.0 - v2.4.3) released over a period of 22 years. 89 versions of samba released over a period of 15 years and grew from 5514 LOC to more than 1,000,000 LOC were analysed. 60 versions of Vsftpd were analysed. 172 versions of SQLite were analysed; starting with its first version v1.0 of 17723 LOC to v3.6.11 of 65108 LOC.

3.2 Metrics and Measurement Tools

Three tools are selected namely Resource Standard Metrics(RSM) and PMCCABE. RSM measures several metrics for C, C++, C# and Java source code files. The metrics which are measured by RSM are total function parameters, total cyclomatic complexity, file function count, total function return, average cyclomatic complexity, average interface complexity; these metrics are used in this empirical study.

We used PMCCABE which is a command line tool in Linux environment. It calculates McCabe-style Cyclomatic Complexity for C/C++ source code, statements in function, lines of code in function, blank lines and the likes. PMCCABE uses per-function metrics; we ran PMCCABE with Unix scripts in the Debian command line . We use metric values from PMCCABE tool for tracing individual functions and their metrics values for analysis purpose because PMCCABE gives a list of all function with their associated various metrics values. The metrics values of the two tools are correlated and this gives us confidence that the metrics values are correct; however we still did visual inspection.

To characterize Increasing complexity property, we used the following metrics:

i. Total Function Cyclomatic Complexity (TF-MCC): this is the summation of the MCC of all the functions in a software

ii. Average Function Cyclomatic Complexity(AF-MCC): this is the average of TF-MCC i.e. $\frac{TF-MCC}{[total\ number\ of\ functions]}$

iii. Total Function Interface Complexity(TF-IC): is the summation of all the Interface Complexity (IC) of all the functions in a software system where the IC of a function is the total number of function parameters and function return statements.

iv. Average Function Interface Complexity (AF-IC): this is the average of TF-IC i.e. $\frac{TF-IC}{[total\ number\ of\ functions]}$

The tools(RSM and PMCCABE) require source code files to parse from a specified source directory and they generated metrics results to specified output folder.

Increasing complexity : as E-type software system evolves its complexity which indicates its deteriorating structure increases unless work is done to maintain or reduce it. When software is evolving, the first concern is the needed functionality. Thus the changes are typically done as a patch, disregarding the integrity of the original design which causes increasing complexity. The implication is that in order to keep the software operational, it is not enough to invest in changing it; one also needs to invest in reducing the complexity repeatedly to acceptable levels. The software systems used as datasets for this project are well evolved software systems, so they are good candidate softwares to quantify increasing complexity. Research findings [16],[17] show that cyclomatic complexity should not be more than the value of 10 per function; else such function is too complex and should be considered for refactoring.

4. Results and Discussion

4.1 Results

Having measured the sample OSS using the measurement tools, the following graphs depict the relationship between the selected metrics under consideration. Figure 1 depicts the behaviour of the Bison OSS. Figure 1 (a)

shows how the TF-MCC and TF-IC grows over the various versions of Bison. Figure 1 (b) depicts the AF-MCC and AF-IC growths over the versions.

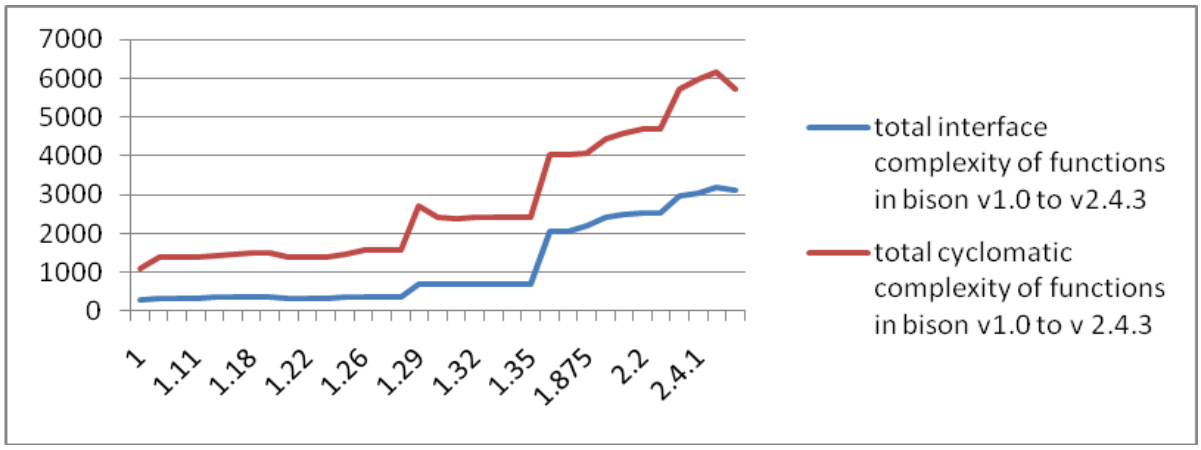

Figure 1(a): total interface complexity and total cyclomatic complexity of functions against the versions of Bison (v1.0 to v2.4.3)

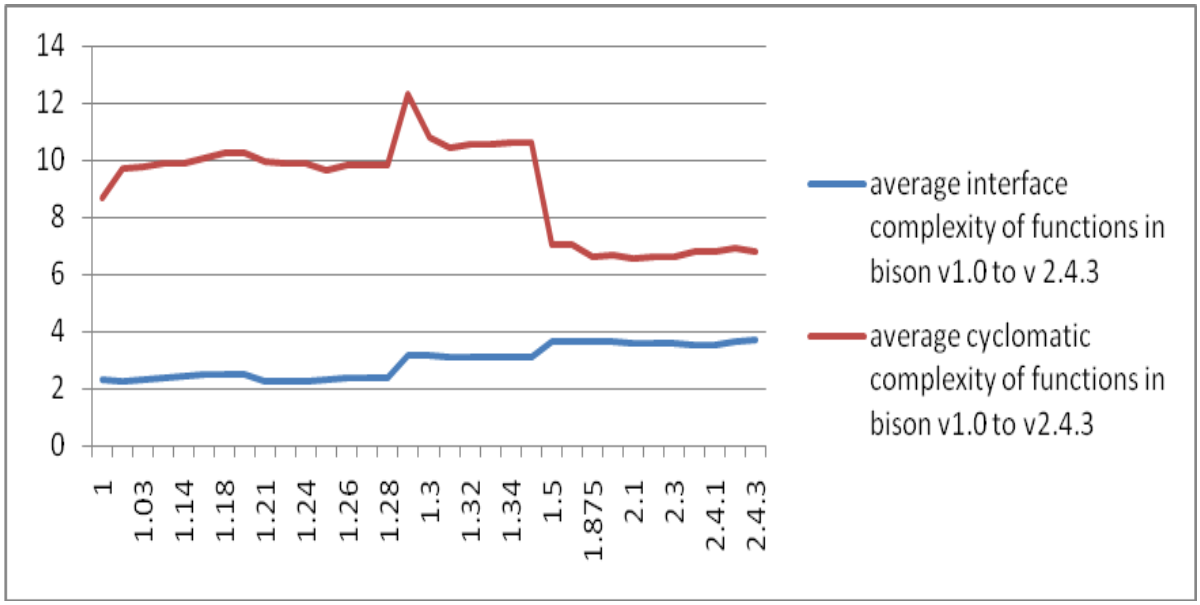

Figure 1(b): Graph of average interface complexity and average cyclomatic complexity of functions against the versions of Bison (v1.0 to v2.4.3)

Figure 1: The complexities and average complexities graph of Bison OSS

Similarly, the graphs for the Samba OSS are shown in Figure 2: Figure 2(a) shows the TF-MCC and TF-IC of the Samba OSS over its versions. Figure 2(b) presents the average MCC and average IC of the OSS.

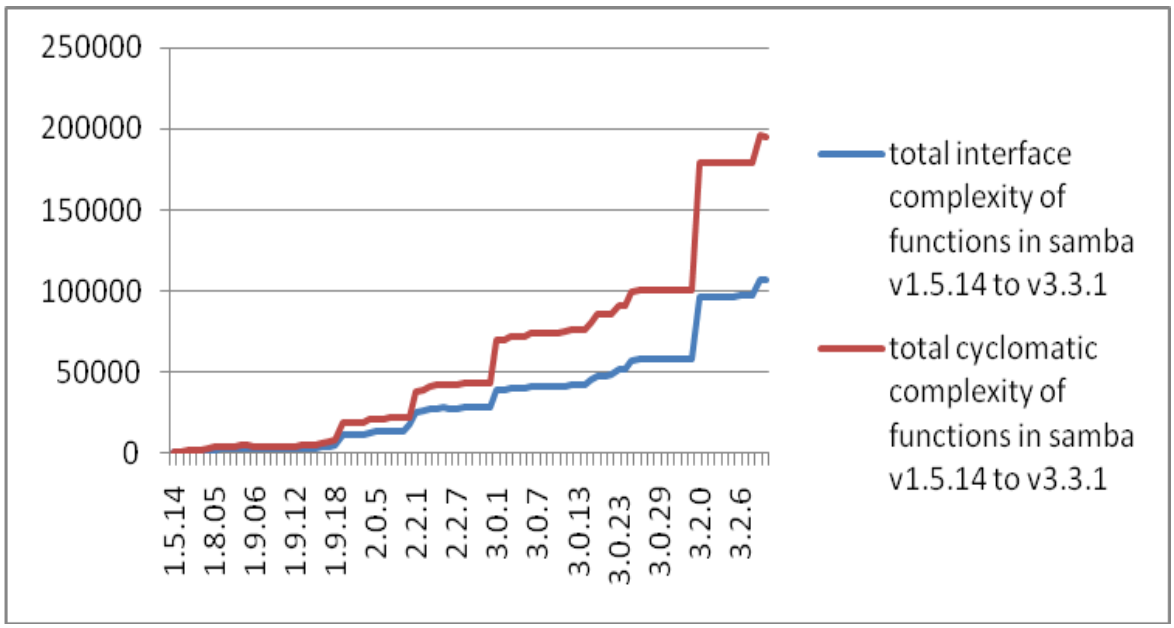

Figure 2(a): Graph of total interface complexity and total cyclomatic complexity of functions against the versions of samba (v1.5.14 to v3.3.1)

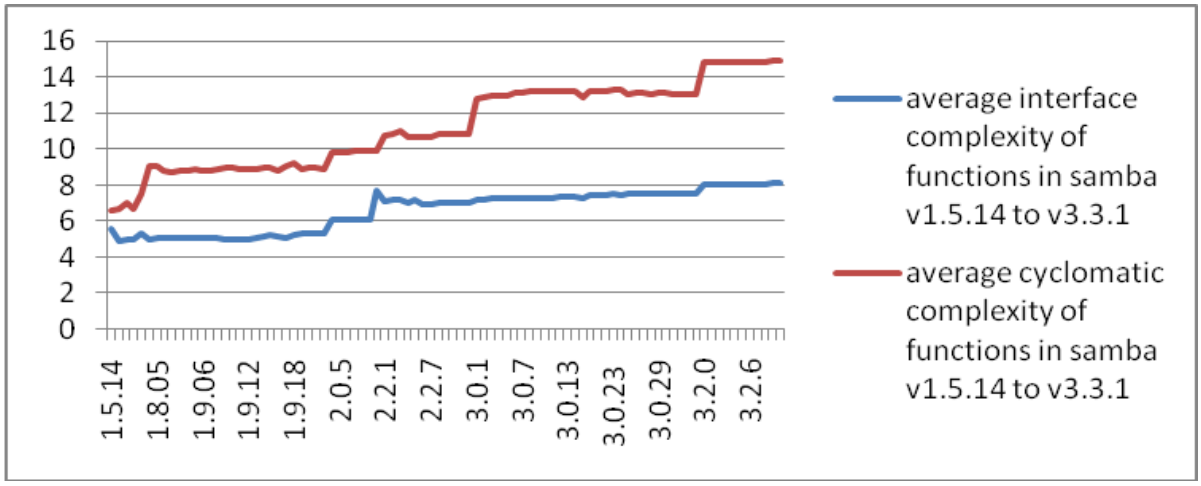

Figure 2(b): Graph of average interface complexity of functions and average cyclomatic complexity of functions against versions of samba (v1.5.14 to v3.3.1)

Figure 2: The complexities and average complexities graph of Samba OSS

The complexities and their averages for the Vsftpd OSS is presented in Figure 3: figure 3(a) shows the TF-MCC and TF-IC while Figure 3(b) shows the AF-MCC and AF-IC of Vsftpd. Figures 3(c) and 3(d) further gives a depiction of the complexities and averages for the development branch of Vsftpd, specifically v1.1 development branch releases. Figures 3(e) and 3(f) also shows the complexity and average complexity measurement over the versions of a stable branch, specifically v2.0 releases. v2.1 was included to show the behaviour when transiting from a stable branch to a development branch.

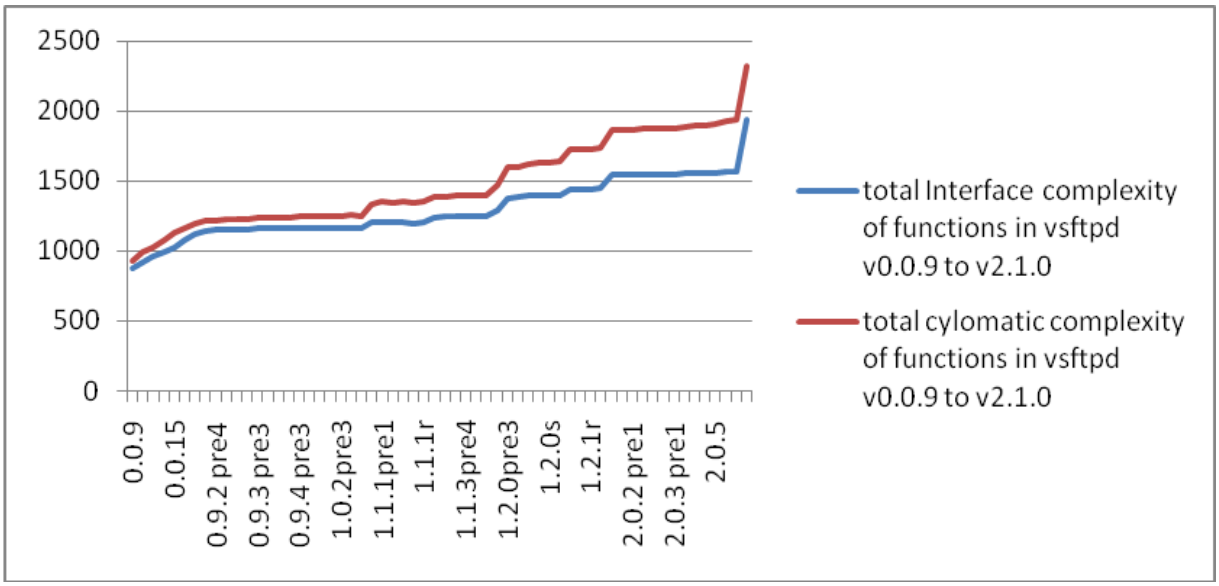

Figure 3(a): Graph of total cyclomatic complexity and total interface complexity of functions against the versions of vsftpd (v0.0.9 to v2.1.0)

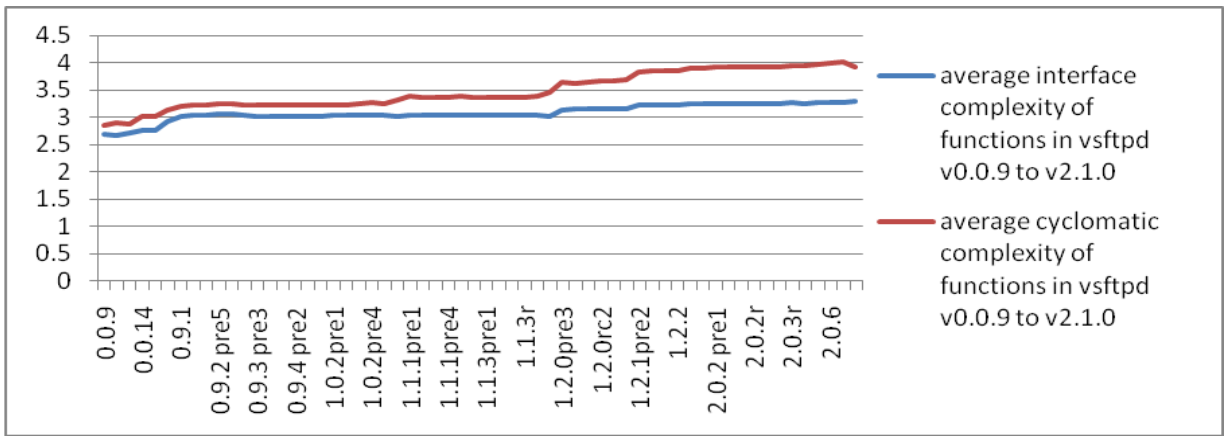

Figure 3(b): Graph of average cyclomatic complexity and average interface complexity of functions against the versions of vsftpd (v0.0.9 to v2.1.0)

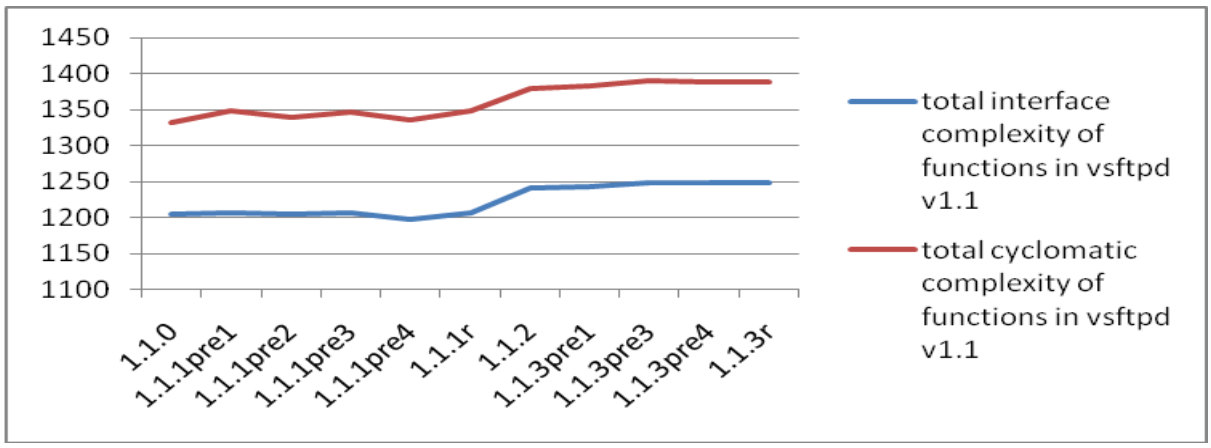

Figure 3(c): Graph of total interface complexity and total cyclomatic complexity of functions against the development versions of vsftpd (version 1.1 releases)

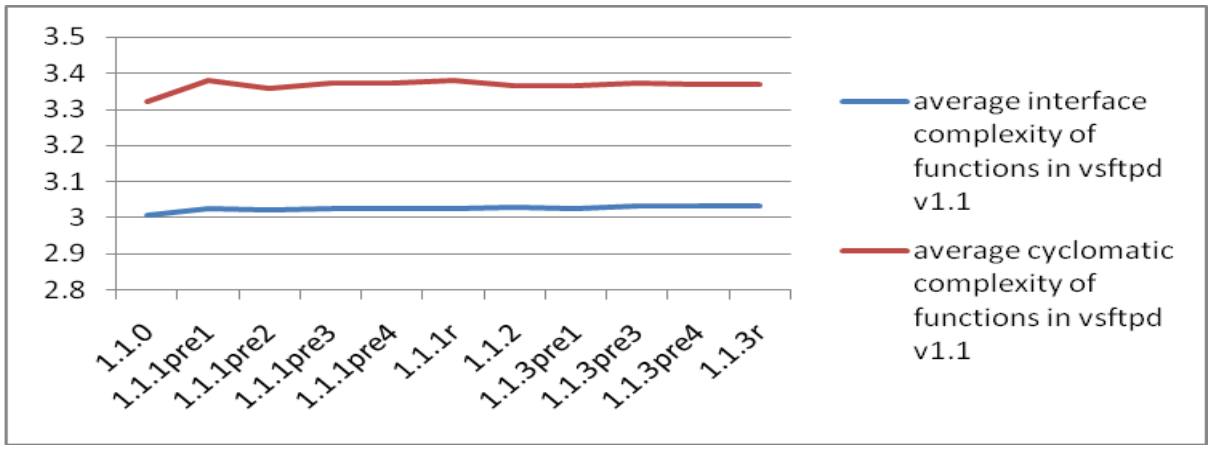

Figure 3(d): Graph of average interface complexity and average cyclomatic complexity of functions against the development versions of vsftpd(version 1.1 releases)

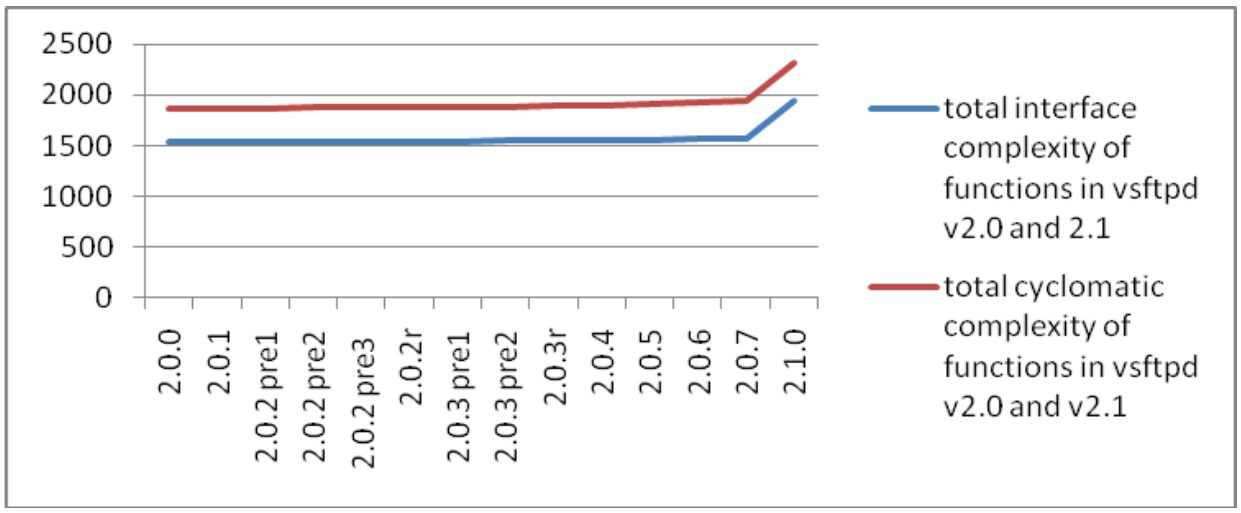

Figure 3(e): Graph of total interface complexity and total cyclomatic complexity of functions against the releases of vsftpd version 2.0 stable branch

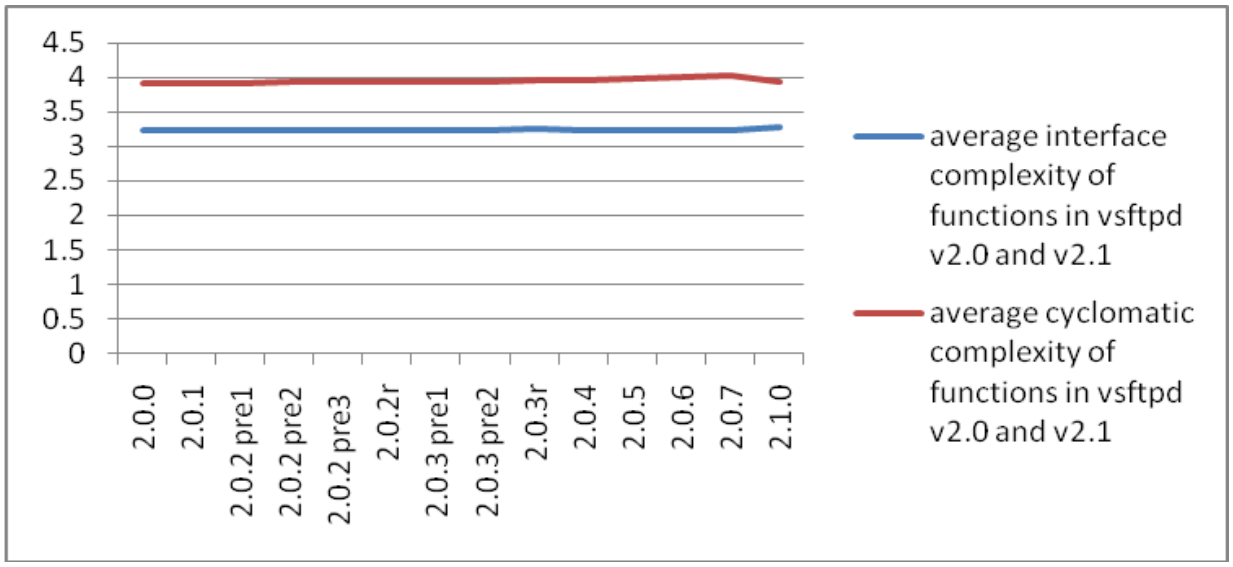

Figure 3(f): Graph of average interface complexity and average cyclomatic complexity of functions against the releases of vsftpd version 2.0 stable branch

Figure 3: The complexities and average complexities graph of Vsftpd OSS

The complexities and their averages for the SQLite OSS is presented in Figure 4: Figure 4(a) shows the TF-MCC and TF-IC while Figure 4(b) shows the AF-MCC and AF-IC of SQLite.

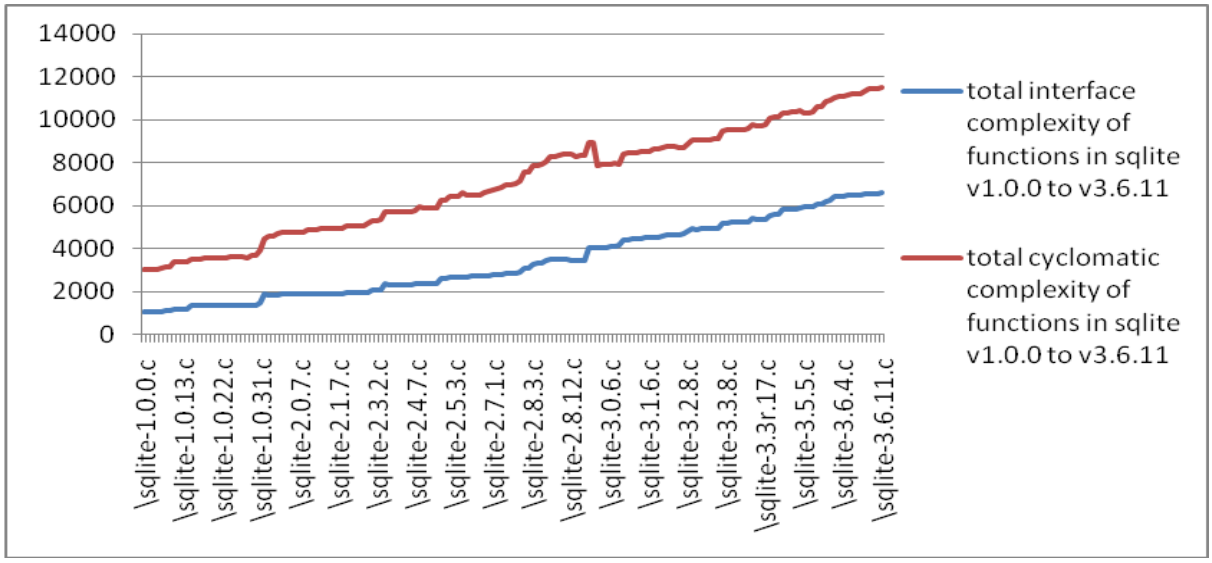

Figure 4(a): Graph of total cyclomatic complexity and total interface complexity of functions against the versions of SQLite

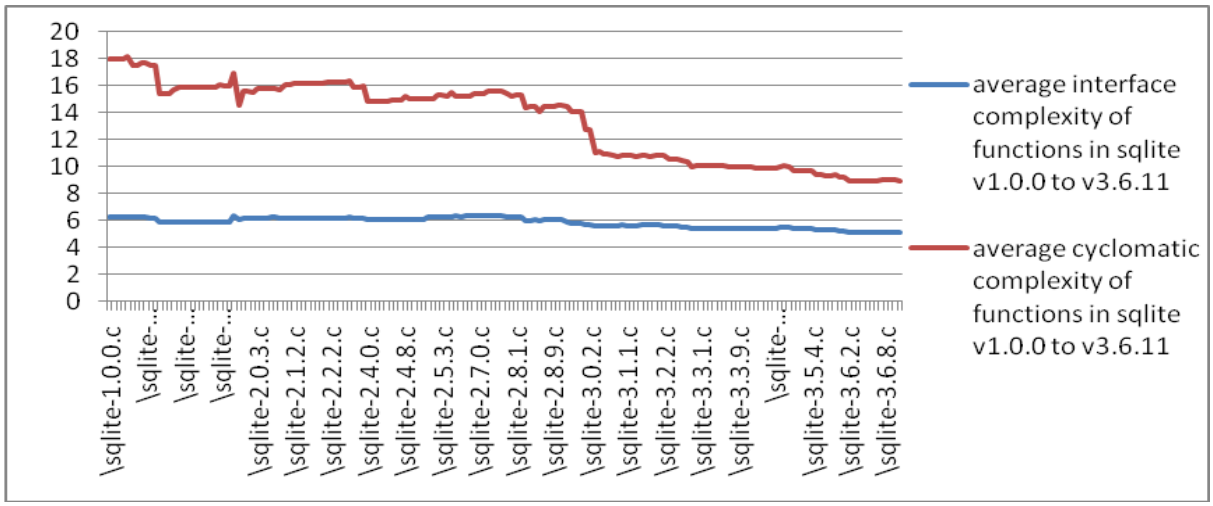

Figure 4(b): Graph of average cyclomatic complexity and average interface complexity of functions against versions of SQLite

Figure 4: The complexities and average complexities graph of SQLite OSS

4.2 Discussion

There is sharp rise in the trend of total interface complexity of Bison, as seen in Figure 1a which means more functions are been added to Bison software system in the course of its evolution. However, the added functions did not really make the system to be too complex as the normalised complexity (the average interface complexity per function) trend in Figure 1b increased within short range. Bison total cyclomatic complexity grew dramatically (Figure 1a) which means that as Bison evolves the function cyclomatic complexity is definitely increasing greatly but work is done to bring the complexity down. The work done is the addiction of relatively small cyclomatic complexity functions; this is why the average cyclomatic complexity declined in the trend of the normalised graph. In the work of [6]on validating Law of Increasing Complexity using Linux Kernel

as case study, they found evidence for the works done to reduce cyclomatic complexity by combination of code improvements and addiction of many low-cyclomatic complexity functions. Similarly, Neamtiu et al. [7] found that the absolute values for cyclomatic complexity and common coupling increase because program size increases, but normalized coupling metric declines while average cyclomatic complexity metrics trends decline or slightly increase. There is evidence of addiction of low complexity functions as the nine OSS used as case studies continue to evolve, Neamtiu et al. [7] suggested that complexity reducing releases are a by-product of large-scale architectural changes or re-engineering. The functions interfaces are not re-factored to provoke vast reduction in the interface complexity like it is in cyclomatic complexity.

The complexity trends for both interface and cyclomatic complexities of Samba software as seen in Figure 11a to 11d grew drastically i.e. Samba interface complexity ranges from 670 to 106918 over the 89 versions while Samba cyclomatic complexity grew from 729 to 195308; but the average interface complexity of the system grew within a short range i.e. 5.5 to 8.12 while the average cyclomatic complexity still grew within a wider range i.e. 6.6 to 14.8. There is proliferation of functions in the evolution of Samba but not of relatively low complexity, this is why the average complexity trends did not decline but grew at a short range or slightly. The functions are not well re-factored to reduce the interface complexity. However the increasing complexity did not deteriorate the system structure to the point of killing the evolution of samba systems. Here the rise in complexity support law of increasing complexity but the fact that it does not affect the successful evolution of Samba software system also refutes the law of increasing complexity from implication point of view. This is one of the reasons why the Law of Increasing complexity can not be validated in some software evolution studies [3], [5], [11], [6], [7].

Top 10 cyclomatic complexity functions were collected in versions of Samba software systems, it is seen that within the pace of 10 releases, about 7 out of the 10 most complex functions still remained the highest and they were seen increasing in complexity but at the same time more low complex functions were been introduced to the software systems. However, after 10 releases, entirely new functions were seen to be the most complex. The former set of highest complex functions were re-factored and given new names and also more functions were introduced to the system which are of higher complexity than the former top complex functions

For the vsftpd software systems, it is seen in Figures 3a and 3b that the total cyclomatic complexity of the softwares from v0.0.9 to v2.1.0 increased within a very large range, from 926 to 2318 while the average cyclomatic complexity of functions increased within a very close range from 2.8 to 4.0 as seen in Figure 3b. The trend of interface complexity was like the cyclomatic complexity because the total interface complexity grew steadily but grew within a close range in average interface complexity trend.

As noted in Figure 3(c) both the total cyclomatic complexity and total interface complexities grew within the major development trend of vsftpd version1.1 while the trends in Figure 3(d) almost levelled off in the average cyclomatic and average interface complexities. Also, Figure 3(e) shows that both the interface and cyclomatic complexity in the stable branch (vsftpd 2.0) seem almost levelled off, unlike in development branch that the complexities increased. A sharp increase in the complexity is observed when the stable branch transits into the development branch. This implies that lots of function were added to the software which increased the complexity but in the average cyclomatic complexity trend, the trend decline which means that the added functions are of low complexity values. The low complexity functions that were added could not control or reduce the interface complexity because it is seen rising in the average interface complexity trend.

The cyclomatic complexity and the interface complexity sharply increase in the evolution of SQLite systems which depict that more functions were been added to the software and thus caused increasing complexity. The average cyclomatic complexity and average interface complexity declined. That means the software is not really increasing in complexity as is evolving, the

cyclomatic complexity has been controlled by addition of lot of low complexity functions to the software system. Also, the interfaces of the added small functions are well re-factored or designed because the interface complexity also declined which rarely occurs when cyclomatic complexity increases.

To ascertain the works done in reducing cyclomatic complexity by addition of many functions with relatively low complexity, a closer look is taken into the metrics report generated by PMCCABE for SQLite using a statistical application function (CountIf) to determine the number of functions with cyclomatic complexity within the range of 1 to 5, and then find the percentage of them, it gave us 55.29% while the same process was done for about the last version of sqlite, namely sqlite v3.6.10. We got 66.5% in this case, thus confirming that more low cyclomatic complexity were added to sqlite software systems in the course of evolution and thus cause the decline in the average cyclomatic complexity of the software.

The interface complexity of software systems in evolution increases as the cyclomatic complexity increases but cyclomatic complexity is being more controlled than interface complexity with addition of lot of less complex functions as the software system evolves. Out of the four software systems that were studied, only in Sqlite software that interface complexity decreases as the software system evolves. In other software systems, interface complexity increased within short range. In two software systems, average cyclomatic complexity declined, it increased within short range in one software system and increased super linearly in Samba softwares evolution. Proactive measure is what can help in controlling interface complexity because multiple exit points which constitute to interface complexity are easy to program but they increase maintenance costs/effort [18]. Functions with greater than 6 input parameters can be very difficult to use on a routine basis and are problematic to parameter ordering. Functions with greater than one return point break the single entry/single exit design constraint which should be the standard unless in cases where it is impossible. Functions of this type are difficult to debug when a runtime error occurs [13].

Generally from the results presented in Fig.1(a) to Fig. 4(b), the total complexity of the sample OSS increases as the software systems evolve over time while the average complexity either increases at a slower rate or decreases. This is glaring in the shape/pattern of the trends in evolution graphs.

5. Threats to Validity

The following are the possible threats to the validity of this study. Firstly, construct validity which is about the accuracy of the metrics used in capturing the system behaviour or characteristics is addressed by using two measurement tools: RSM and PMCCABE. The results of the measurements from the two tools were compared to ensure that cyclomatic complexity and interface complexity metrics are the same from the tools. Content validity is also reduced because the datasets used as case studies are official releases; they are not individual commit which may not make it into official release.

Internal validity connotes the causal relationship between the dependent and independent variables. It ensures that the changes in the dependent variable can be attributed to changes in the independent variable. This study investigated the trend in the complexity (the dependent variable) of OSS over several versions (the independent variable) of the OSS. Thus, it is obvious that the change in complexity has causal relationship with the versions of the sample OSS used in this study; this is depicted by Figures 1 – 4 in the result section.

External validity is the extent to which the results of this study can be generalised. The investigation conducted here uses systems developed in C. Thus, the results may not hold for other systems developed using Object-Oriented (OO) methodology since the sample OSS are absolutely procedural and it is believed that Object-Orientation reduces complexity. More studies to investigate the Lehman's laws in the context of OO systems need to be carried out to test the validity of the findings of this study. Nevertheless, going by the findings reported by [4] that software systems are self-similar, the findings of this investigation will most probably hold in other software context.

6. Conclusion

The validity of Lehman's second law of software evolution for OSS developed using the procedural C programming language is investigated. The followings are the findings of the study.

Total cyclomatic complexity of systems increases within a very large range in the course of evolution, while the average cyclomatic complexity of functions is increasing within a very close range or declines as the case is in vsftpd systems. The trend of interface complexity is similar to the cyclomatic complexity because the total interface complexity grew steadily but grew within a close range in average interface complexity trends. The total cyclomatic complexity and total interface complexity often grow sharply within the major development trends while the trends either almost levelled off in the average cyclomatic and average interface complexities. The interface and cyclomatic complexity within stable branch in most cases seem almost levelled off or grow linearly, while in development branch the growth is more or super linear. Increase in the number of functions in system is proportional to increase in the magnitude of total cyclomatic complexity and total interface complexity in such system.

Also when the trend of stable release is crossing to a development major release, there is usually a sharp increase in the two complexity metrics, which means lot of functions are added to the system which increased the complexity but in the average cyclomatic complexity trend, the trend decline which means the added functions are of low complexity values. The low complexity functions that were added could not control or reduce the interface complexity because its trend is seen rising in most cases.

The interface complexity of systems in evolution increases as the cyclomatic complexity increases but cyclomatic complexity is being more controlled than interface complexity with addition of lot of less complex functions as the system evolves, out of the four systems that were studied so far just in sqlite that interface complexity was able to be brought down to declining state in evolution; in others it increased within short range; but in two systems average cyclomatic complexity declined, it increased within short range in one system and continuing to increase super linearly in samba systems. Proactive measure is what can help in controlling interface complexity because multiple exit points which constitute to interface complexity are easy to program but they increase maintenance costs/effort. Functions with greater than 6 input parameters can be very difficult to use on a routine basis and are problematic to parameter ordering. Functions with greater than one return point break the single entry/single exit design constraint which should be the standard unless in cases where it is impossible. Functions of this type are difficult to debug when a runtime error occurs.

When an E-type system is increasingly changed, the overall complexity increases i.e. by summing up the complexities of functions in the software. However by normalized metric like per function, it is noted that the complexity may not really be increasing which is the case in bison, vsftpd, samba and sqlite systems. Therefore, the law of increasing complexity cannot be confirmed by this study. However the law is not completely refuted given that it stated that complexity increases except work is done to check it. It is most likely that the developers of the sample OSS put in effort to keep complexity in check as the systems evolve. Thus, there is no substantial increase in the average complexities of each of the systems studied.

Finally, to ensure the external validity of the findings of this study on Lehman's second law, more studies using Object Oriented software sample will be carried out. Also, the number of sample software and the number of complexity metrics will be increased. Metrics such as Response For Class (RFC), Weighted Method for Class (WMC) and coupling are worth considering in future works.

References

[1] M.M. Lehman, "Laws of software evolution revisited", *In European Workshop on Software Process Technology, Springer Berlin Heidelberg.* pp.108-124, 1996

[2] M.M. Lehman, "Programs, life cycles, and laws of software evolution", *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060-1076, 1980.

[3] M.W. Godfrey and Q. Tu, "Evolution in open source software: A case study", *In Software Maintenance, 2000.Proceedings of IEEE International Conference on Software Maintenance,2000,* pp. 131-142

[4] I. Herraiz, "A Statistical Examination of the Evolution and Properties of Libre Software, PhD thesis*", Universidad Rey Juan Carlos*, 2008, Retrieved from *http://purl.org/net/who/iht/phd.*

[5] J. Fernandez-Ramil, A. Lozano, M. Wermelinger, and A. Capiluppi, "Empirical studies of open source evolution", *In Software evolution ,Springer Berlin Heidelberg,*2008, (pp. 263-288)

[6] A. Israeli and D.G. Feitelson, "The Linux kernel as a case study in software evolution", *Journal of Systems and Software*, Vol.*83, No.*3, 2010, pp.485-501.

[7] I. Neamtiu, G. Xie, and J.Chen, "Towards a better understanding of software evolution: an empirical study on open source software", *Journal of Software: Evolution and Process*, Vol. 25,No.3,2013, pp.193-218.

[8] S. Pirzada, "A statistical examination of the evolution of the UNIX system (Doctoral dissertation)", *University of London,* UK, 1988.

[9] M. Alenezi and K. Almustafa, "Empirical analysis of the complexity evolution in open-source software systems", *International Journal of Hybrid Information Technology*, Vol. 8, No.2, 2015, pp. 257-266.

[10] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data", *IEEE Transactions on Software Engineering*, Vol. 27, No.1, 2001, pp. 1-12.

[11] R. Vasa, Growth and change dynamics in open source software systems, Ph.D. Dissertation, Swinburne University of Technology, Melbourne, Australia,2010.

[12] M. K. Debbarma, S. Debbarma, N. Debbarma, K. Chakma, and A. Jamatia, "A Review and Analysis of Software Complexity Metrics in Structural Testing," *Internal Journal of Computer and Communication Engineering*, vol. 2, no. 2, pp. 129–133, 2013.

[13] M Squared Technology LLC, Metrics Definitions*,2009,* Retrieved August 7, 2016 from *http://www.msquaredtechnologies.com/m2rsm/docs/rsm.metrics-narration.htm*

[14] C.K. Roy and J.R. Cordy, "Evaluating the evolution of small scale open source software systems", *Special Issue: Advances in Computer Science and Engineering,* 2006,pp.123.

[15] D.I. Sjøberg, T. Dybå, B.C. Anda, and J.E. Hannay, "Building theories in software engineering", *In Guide to advanced empirical software engineering, Springer London,* 2008, pp. 312-336

[16] A.H. Watson, D.R. Wallace, and T.J. McCabe, (1996), "Structured testing: A testing methodology using the cyclomatic complexity metric", US Department of Commerce, Technology Administration, National Institute of Standards and Technology, Vol. 500, No. 235, 1996.

[17] A. Khannur, *Software Testing: Techniques and Applications*, Pearson Education India, 2011.

[18] G.W. Lecky-Thompson, *Corporate Software Project Management (Charles River Media Computer Engineering)*, Charles River Media, Inc., 2005.