I. J. Engineering and Manufacturing, **2023, 4, 20-33**
Published Online on August 8, 2023 by MECS Press (http://www.mecs-press.org/)
DOI: 10.5815/ijem.2023.04.03

Software Evolution of Next.js and Angular

Anika Tabassum*
Institute of Information Technology, University of Dhaka, Bangladesh
Email: bsse0925@iit.du.ac.bd
ORCID iD: https://orcid.org/0009-0004-6830-3726
*Corresponding Author

Ishrat Jahan Emu
Institute of Information Technology, University of Dhaka, Bangladesh
Email: bsse0927@iit.du.ac.bd
ORCID iD: https://orcid.org/0009-0005-5801-6464

Abdus Satter
Institute of Information Technology, University of Dhaka, Bangladesh
Email: abdus.satter@iit.du.ac.bd
ORCID iD: https://orcid.org/0000-0002-2053-7150

Received: 16 January, 2023; Revised: 04 March, 2023; Accepted: 27 April, 2023; Published: 08 August, 2023

Abstract: Software evolution is the process of updating a software from it's initial phase incorporating updates taken from the users. E-type software takes feedback and requirements from the users and updates the software accordingly. In the 70's, Lehman proposed some laws of software evolution, which are still considered as the standards of measuring software evolution. Now-a-days, E-type softwares are getting very popular as users now use softwares by giving feedback. We are measuring Lehman's laws of software evolution for two E-type softwares, Next.js and Angular. By downloading the source codes from github repositories, we analysed the source codes and evaluated if the softwares follow Lehman's Laws of Software Evolution or not. This analysis completes the research which aims at measuring if the softwares evolve by maintaining Lehman's Laws.

Index Terms: Software Evolution, Lehman's Laws, Maintenance, Software Quality

1. Introduction

Software evolution refers to the modifications of a software system over time, performed by software maintenance. It studies how dynamic systems change over time. Software evolves with user requirements and feedback. There are three types of software, P-type, S-type and E-type software. Among them E-type software are the real world software which works with collecting feedback from users and updating the software as per the feedback. In recent years E-type software has gained popularity because of high usability. The more software is maintainable the more it is usable. Software evolution analyzes E-type systems to evaluate a system's maintainability. If an E-type system is not adaptable, the users will lose interest using it which will cause failure to the software. To understand the evolution and maintainability of software, it needs to be analyzed properly. Lehman has proposed eight laws of software evolution to measure maintainability of the software.

E-type software is getting popular by users now-a-days. By users, we mean developers here. In this era of digitizing, almost every company has a website. Millions of developers are working to build website for various companies. In the process of web development, there are front-end development and beck-end development. There are several frameworks which are rated highly by the developers. Javascript has been a very popular language and been used for web development as well as web based teaching approach. [1] We chose two of those front-end framework named Next.js and Angular, written in Javascript. Both of them are open source, collects feedback from the users and update the software to get more usability. As they are popular from the start and still being highly used by the developers. Next.js is being used by popular tech giants like Uber, Netflix, Starbucks, we are interested in evaluating how well these systems are maintained.

As the high use of front-end frameworks, which are E-type software, this software should follow the laws of software evolution. We are trying to analyse the source code from the repository, Github, where the codes are kept.

Github is also used for getting feedback from the community. So by analysing github source code, we will connect the developers' feedback with the update of the software systems and we will measure the laws given by Lehman. As the continuous growth of E-type softwares, we wanted to measure two E-type systems, Next.js and Angular to evaluate maintenance as per Lehman's Law. No studies have been conducted so far on these two E-type software systems in this manner. Previous works have been done on measuring software evolution on linux kernel [2] using similar manner to ours. The details about the existing solutions in different areas are described in the next chapter.

The goal of software re-engineering is software maintainability. Software modifiability is crucial to software maintenance since it determines how simple it will be to remedy component problems and adapt to a new environment[3]. The objective of this research is to evaluate software maintenance by analysing the source codes. We hope to follow the working methodology of [2] and fulfil our research objective. For evaluating software maintenance, we extracted the source code of the systems from github and analysed the source code with software analysing tool. Then from the results of the tool, we plotted regression line to visualize if the systems follow the Lehman's law or not.

After our evaluation we found that, Next.js follows all 6 laws that we are working with. But for Angular, it did not follow any of the laws of Lehman. This finding is surprising as Angular is a very popular and well accepted E-type software system. It is maintaining it's popularity but not following the Lehman's laws of software evolution.

In the next sections the related works, Case Study, the results and discussions are given.

2. Literature Review

Analysis of software evolution is one of the most well researched elements of software development and maintenance. This type of empirical study is considerably aided by the availability of multiple data in software repositories, which enables the investigation of research issues concerning all aspects of a software project, including its source code, documentation, developers, bug reports, and so on. Since the 1970s, software evolution has been a subject of research.

Lehman first defined three fundamental principles of software evolution in 1974 [4], based on his analysis of the OS/360 operating system. Lehman later updated the previous principles and proposed two new ones [5]. Lehman published a revised version of laws III, IV, and V [6] in the early 1980s. Lastly, Lehman produced a new edition of the statutes that included new provisions [7] and republished the most recent versions in 2006 [8]. The laws by Lehman introduced the softwares with some ground truth of evolution. These are the building blocks and have been used as starting point of measuring software evolution.

Many researchers have published their work by analysing different types of E-type systems. Some evaluated open-source mobile applications [9,10], some evaluated operating systems and browsers [11], some evaluated linux kernel system [12]. Many of those E-type systems abide by the Lehman's laws of software evolution. Many could not proof all the laws because of not getting all the metrics to measure the laws. Likewise the literature, we will work but with a new E-type software system, front-end frameworks.

In the paper [9], Li et al. analysed some android applications and tried to measure software evolution like Zhang in [10]. This work is very relevant to ours but in a different field. Lone et al. analysed MacOS and LinuxOS and tried to evaluate operating systems. But as the worked with operating systems, they have used some other metrics than Lehman used in his ground study. They analysed monthly use, yearly use, cumulative use to figure out the growth of the operating systems and browsers. This study was a great kickstart for the analysis of that kind of softwares. In the paper [13] Newhook measured evolution of mobile application, but it was from design perspective. We tried to measure evolution from source code, so this literature was not so much aligned with ours. The authors of [14] worked with glibc and measured software evolution. This study was inspiring to analyse our work from using github repository.

But mostly, the evolution of linux kernel system [12] has been the most inspiring work for us. Though the systems we chose to analyse were not like linux kernel system, it was E-type software system and very fulfilled study on software evolution. So we followed the literature and incorporated with new E-type system softwares.

3. Methodology

The case study was designed in accordance with Runeson and Host's (2008) principles. We analyzed the development of Next.js and Angular framework using Lehman's laws, which describe software quality trends and changes.

This study aims to examine the changes and quality of the Next.js and Angular frameworks through time using Lehman's laws of evolution. We followed the following steps to complete the study. Since we focused on changes between versions and their impact on quality, the remaining two laws were outside our scope. The processes we followed are –Making Research Questions, Data collection and processing and Statistical Analysis. We addressed the following research questions

A. Research Questions

RQ1: Is Law I, "Continuing Change" has been confirmed by the Next.js and Angular frameworks. Does this m6

RQ3: Is Law III's "Self-Regulation" supported by the Next.js and Angular frameworks as a sign of a trend in changes?

In this question, we'll investigate whether the Next.js and Angular frameworks' total number of modified files are satisfactory or not.

RQ4: Is Law IV's "Conservation of Organizational Stability" validated by Next.js and Angular as an indicator of change?

In this question, we'll investigate whether Next.js and Angular frameworks releases provide consistent work.

RQ5: Is Law V referring to "Conservation of Organizational Stability" validated by Next.js and Angular frameworks as an indicator of a changing trend?

In this question, we will determine if the new content added between releases of Next.js and Angular frameworks is stable or if there are significant variations.

RQ6: Is Law VII about "Declining Quality" confirmed by the Next.js and Angular frameworks as an indicator of a quality trend?

In this research question, we want to identify if the quality of a software system gets worse over its lifecycle.

B. Data Collection and Preprocessing

For data collection we used github repositories. We used Next.js and Angular repositories of github*. We used python to code and fetched all the source code folders along with the source codes which are written in typescript mainly. There are also javascipt files which were also analysed. We downloaded the versions till 6th November, 2022. The repositories downloaded as zip file. So we extracted all the files and analysed it using understand tool** by scitools. It is a powerful static code analysis tool. Then we used these data for further statistical analysis. There have been many other studies conducted to using this tool to get the metrics by analysing the source codes. [15] used this tools to get metrics and got 96% accuracy in vulnerability detection. Study [16] also used this tool to analyse static code for finding bad code smells and achieved over 99% accuracy. All these studies have used the same metrics like Line of Codes, Cyclomatic Complexity etc. as ours. These metrics are used for further discussion using python libraries. So the data analysed to further metrics and for statistical analysis are reliable and accurate in this regard.

C. Statistical Analysis

For statistical analysis, we used the data previously mentioned and plotted to find out trends in the releases. For example, we plotted number of files over versions to find out if number of files increases version wise. Likewise, line of code, cyclomatic complexity, number of releases, pre-releases etc. were all plotted against versions to do statistical analysis to find out if the systems follow Lehman's laws or not.

4. Discussion

A. Angular Release Trends

Angular is a component based framework built on typescript. It is used for web application development. Angular framework has components, templates, directives and dependency injection. Version numbers for Angular show the extent of changes made by the release. Three components make up an Angular version number: major.minor.patch. Version 6.4.13, for instance, denotes patch level 13, major version 6, and minor version 4.

From the documentation of Angular, [17] we can see that the levels of changes mean different things in Angular. Such as, the major releases contain major new features. The minor releases contain new but comparatively smaller features. They are fully backward compatible. The patch releases the patch releases are generally bug fix releases which have low risks. There are two types of pre-releases for each major and minor release. The "Next" releases are developing and testing actively. An example of a Next pre-release is 7.4.0-next.3. The release candidate means final testing is being done on the feature. An example of Release Candidate is 7.4.0-rc.3.

Angular follows the release cycle like the following- A major release is built every 6 months. For every major release, there are 1-3 minor releases. Weekly, a patch release and prerelease is built.

For our analysis, we will work with the stable versions and the pre-releases as well.

B. Next.js Release Trends

Next.js is based on typescript language. From the releases of Next.js, we find that the versioning of Next.js is similar to the versioning of Angular.

Like Angular, Next.js has some pre-release versions named canary and beta. The canary version of Next.js releases daily. It has fixes and new features, but those are not added to the stable version yet. So the canary versions are like public beta.

The beta version of Next.js also ships the prereleases. Recently, beta releases are obsolete. Next.js releases features and bug fixes in the canary versions. So we are only working with the stable and the canary versions of Next.js, dropping the beta versions.

C. Lehman's Law:

Lehman proposed eight laws of software evolution by studying several large scale systems. We tried to quantify the laws in the case of javascript language based frameworks (Angular and Next.js). The laws are described below:

1) Continuing Change (1974): "An E-type system must be continually adapted, else it becomes progressively less satisfactory in use." A software never dies if it is useful. The resources of the software are reused. To measure this, we need to measure the LOC and counting files. In Angular and Next.js, the changes are built in the pre-releases. So observing changes in line of code and number of files will help to find answers about continuing change.

2) Increasing Complexity (1974): "As an E-type system is changed its complexity increases and becomes more difficult to evolve unless work is done to maintain and reduce the complexity." It is very hard to prove this law. If complexity increases it proves the initial part of the law. If complexity decreases, it proves the later part of the law. So, this law is hard to prove for java-script applications. Although, we provided the graph by plotting the cyclomatic complexity for all nested functions.

3) Self-Regulation (1974): "Global E-type system evolution if feedback regulated." The growth trend should be observed. In some dataset, the growth trend had been different for major and minor releases. The incremental growth of any particular version by observing the number of files can be measured.

4) Conservation of organizational stability (invariant work rate) (1980): "The work rate of an organization evolving an E-type software system tends to be constant over the operational lifetime of that system or phases of that lifetime." To measure this, we need to get data about the average man hour or number of developers. We want to measure the average effective global rate of activity. Although it is hard to get the number of developers, we can measure the number of releases per year which shows an indication of organizational stability.

5) Conservation of familiarity (1980): "In general, the incremental growth (growth rate trend) of E-type systems is constrained by the need to maintain familiarity". This law says that, two subsequent releases has limited amount of change so that the users and the developers are familiar with the change. To measure this, line of code, number of files, cyclomatic complexity of per functions can be measured. To understand this law from the user's perspective, we will only look into the stable versions because stable versions are intended for users. We will both look into the major and minor versions to understand if Angular and Next.js conserves familiarity or not.

6) Continuing growth (1980): "The functional capability of E-type systems must be continually enhanced to maintain user satisfaction over system lifetime." Growth can be measured by the size of the software product. Software size of the product can be measured by calculating the number of modules and observing the trends. To measure the size metrics, we can measure Line of Code and number of functions in all directories. For Angular and Next.js we could not find any way to measure the functionality metrics.

7) Declining quality (1996): "Unless rigorously adapted and evolved to take into account changes in the operational environment, the quality of an E-type system will appear to be declining." This law is related to the second law. Like the second it is hard to prove as measuring quality itself is hard. To observe the perceived quality, we need to focus on the user acceptance of the frameworks. If the frameworks are used by the users, then it shows that the quality is not declining. To observe the measured quality, the Maintainability Index is measured by measuring LOC, MCC and HV.

8) Feedback system (1974/1996): "E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems." Developers, managers and users are involved in the development system. The stability of growth models are helpful to measure the sizes accurately.

D. Evolution of the Next.js

In this section, we examine Lehman's laws of software evolution in light of data corresponding to the evolution of the Next.js framework. In contrast, Lehman's own research finds no evidence for these laws (Lehman et al., 1997, 1998a,b). Please take note that the laws are often numbered according to the year in which they were first enacted. Laws are rearranged here to follow a new sequence, with those closest to the code showing first.

• Law 1: Continuing change

According to this law, a program must be continuously modified to changes; otherwise, its performance degrades. Generally, it is difficult to distinguish between environmental adaptation and general growth (as expressed by the law of ongoing growth) (Lehman et al., 1998b).

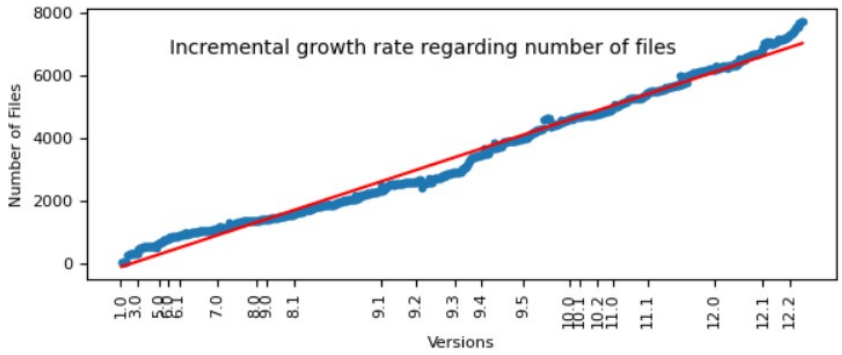

Fig. 1. The growth of number of files

Fig. 1 depicts a graph of how the number of files increases over versions. In the X-axis we plotted the versions from version 1.0.1 to version 12 and in the Y-axis number of files have been plotted. We fitted a regression line to understand the changes in the number of files over versions. As, Next.js started its journey from 2016, there are many releases over time. We found the coefficient as 6.22 and intercept as -111.55. As the coefficient value is positive, we can decide that, Next.js has linear growth over versions.

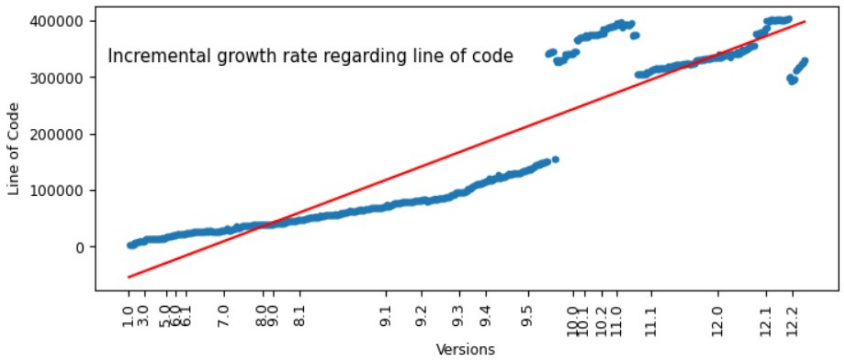

Fig. 2. The growth of number of line of Code

To do further analysis, we have plotted lines of codes over versions and found the same type of linear growth with coefficient value 394.42 and intercept value of -54598.69. (Fig. 2)We can therefore declare that the following program displays continual evolution and adaptation to its surroundings.

• Law 2: Increasing complexity

According to this law, the complexity of a program will rise as it evolves unless maintenance efforts are made to lower its complexity. Formally, this law is difficult to prove or reject since it allows for both trends: if complexity increases, it fits the basic assumption of the law, but if it decreases, it may be due to efforts to reduce complexity, thus also fitting the law. In addition, work on extending the system often entails effort on maintaining its maintainability, and the two cannot be separated in a practical sense (Lawrence, 1982). This rule is supported by Lehman's rationalization (adding features and devices inevitably increases complexity (Lehman, 1980b)) and by data indicating that growth rates fall over time, as would be anticipated due to the limits of greater complexity (Lehman et al., 1997, 1998b).

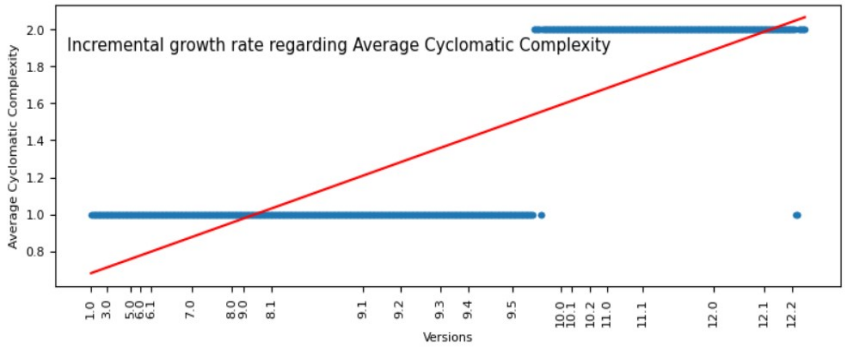

Fig. 3. Cyclomatic Complexity per function Next.js

Fig. 3 shows the linear growth of complexity per function with coefficient value of 185.4 and intercept value of -36319.43. In the X-axis we plotted the versions and in the Y-axis the number of cyclomatic complexity per function has been plotted. As the coefficient value is positive, it shows that cyclomatic complexity increases over time. From the figure, it is clear that at the very beginning the average cyclomatic complexity is increasing, then it decreases and increases too. It is natural because, after collecting the feedback there may be several functions which are complex to implement and increases cyclomatic complexity.

• Law 3: Self-regulation

This law states that the process of program evolution is self-regulating, resulting in a stable trend. Lehman finds evidence for this concept in the fact that empirical growth curves have a ripple superimposed on a continuous growth trend, and argues that the ripple illustrates the interaction between the opposing forces of desired growth and limited resources (Lehman et al., 1997, 1998b). The existence of self-regulation can be determined by observing growth trends, which indicate that deviations from a consistently smooth growth rate will be remedied. The Next.js dataset we utilize has significantly more releases than Lehman's datasets.

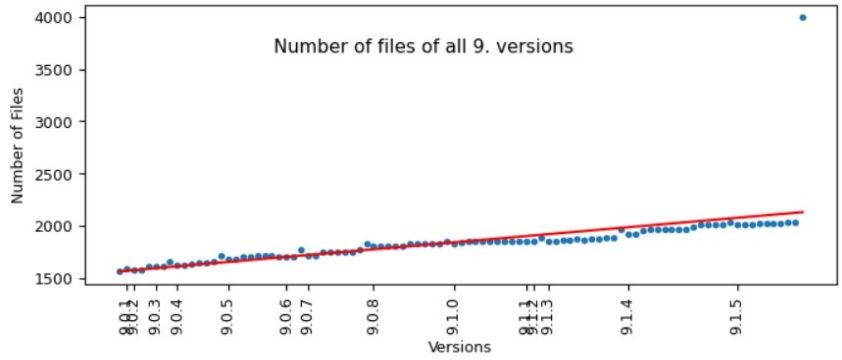

Fig. 4. Number of files all 9 versions

Fig. 4 depicts a graph of how the number of files increases over a year in stable releases of all 9 versions. In the X-axis we plotted the version number and in the Y-axis the number of files of all 9 versions have been plotted. We fitted a linear regression line to find out the relation. The coefficient value is 6.02 with an intercept of 1557, 53 which shows a positive relation between number of files and all the 9.X versions. Typically, the observed development patterns are relatively stable, however there are tiny fluctuations that can be characterized as ripples. Occasionally, though, they display bigger jumps as a result of incorporating an outside designed additional subsystem. The relatively smooth growth could be attributed to self-regulation, or it could be the result of a constant labor rate. The outcome is qualitatively comparable to Lehman's findings, namely that the growth rate appears to oscillate about a mean and that relatively strong growth is almost always followed by sub-average growth, indicating an alternating between growth and stabilization. Consequently, our observations can be interpreted as indirect evidence for the existence of self-regulation.

• Law 4: Conservation of organizational stability (invariant work rate)

This law states that the average effective global rate of activity on an emerging system remains constant throughout the product's lifespan. This statistic is technically troublesome due to the fact that we are attempting to evaluate "effort" on the project. It is difficult to obtain accurate information regarding man-hours or the number of

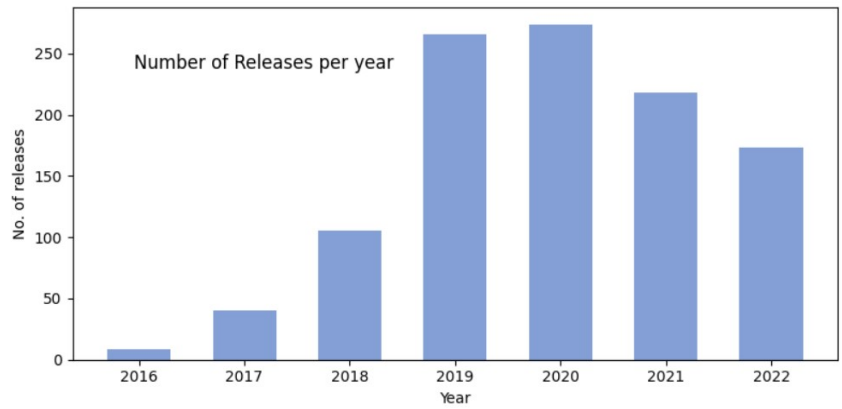

Fig. 5. Number of releases per year

developers in closed-source systems, and even more difficult (and possibly ill-defined) in open-source initiatives. Furthermore, man-hours are frequently incorrect as a metric of work (Brooks, 1975). Lehman advocates using the number of items handled (that is, added, deleted, or modified) as a proxy, but notes that this, too, has methodological challenges (Lehman et al., 1998b).

Figure 5 depicts the number of annual releases from 2016 to 2022. The highest number of releases is given in 2020.

Law 5: Conservation of familiarity

According to this regulation, the changes between successive releases are limited so that both developers and users can keep their knowledge with the code and the system, respectively. Lehman et al. (1997) recommend examining the incremental growth; if it is steady or dropping on average, this shows that familiarity is being conserved. In addition, they propose a threshold that, if two or more consecutive points surpass it, the subsequent points should be near to zero or negative. Lawrence asserts that the series of gradual changes he observed in the examined systems was random and takes this as evidence of a lack of conservation (Lawrence, 1982).

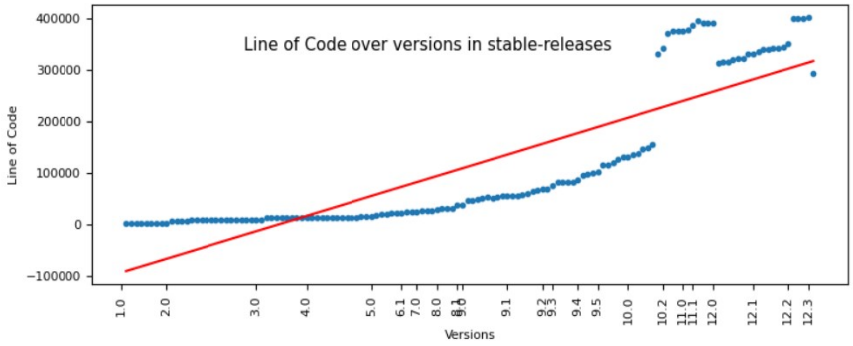

Fig. 6. Line of Code in Stable Releases

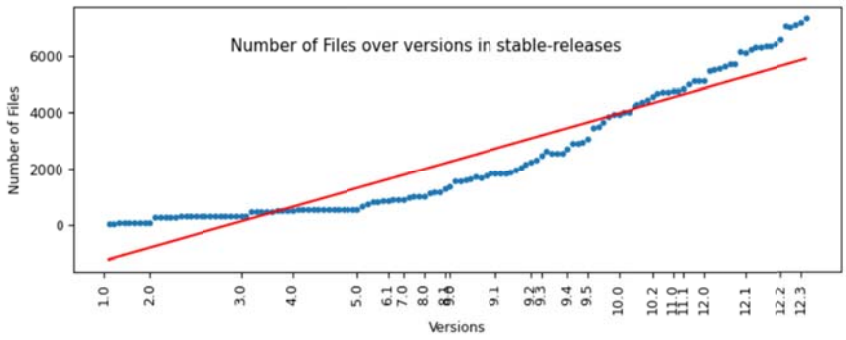

Fig. 7. Number of Functions in Stable Releases

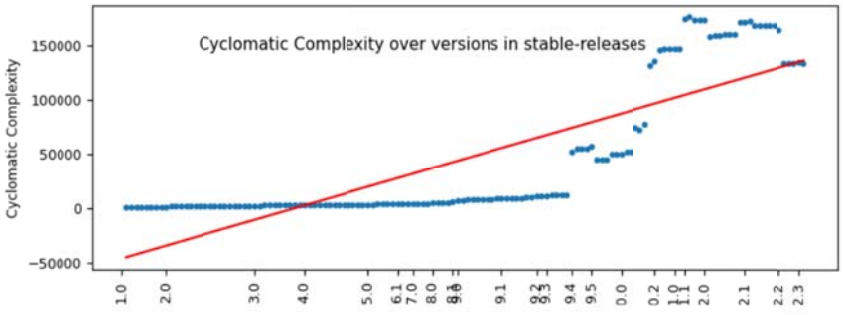

Fig. 8. Cyclomatic Complexity in Stable Releases

However, the maximum change that happened between subsequent versions may be a better test. When we look into the stable versions and analyze the line of code, cyclomatic complexity and number of files, we can see that, Next.js conserves the familiarity. In the case of number of files in the stable versions, we get the coefficient value of 51, 87 which is positive and shows positive relationship in number of files ad versions. For line of code and cyclomatic complexity we got the coefficient value of 1972.99 and 1320, 71. As both of them are positive, we can conclude that, Next.js conserves the fifth law of Lehman.

• Law 6: Continuing Growth

According to this concept, the functional content of a program must be continuously increased during its lifetime in order to maintain user satisfaction. ”Growth” can also be interpreted as referring to the software product’s basic size.

In addition, the size may be used as a proxy for functional content if it is assumed that additional code is developed to support new features. Calculating software size measures (such as the number of modules) and tabulating their patterns over time can thereby validate continued expansion. This was the strategy employed by Lehman and others. These open source projects collect feedback from the users and incorporate those changes. The changes and made and built on the pre-release versions. So by analysing changes on those pre-release versions we can evaluate if the growth of the softwares are continuing or not.

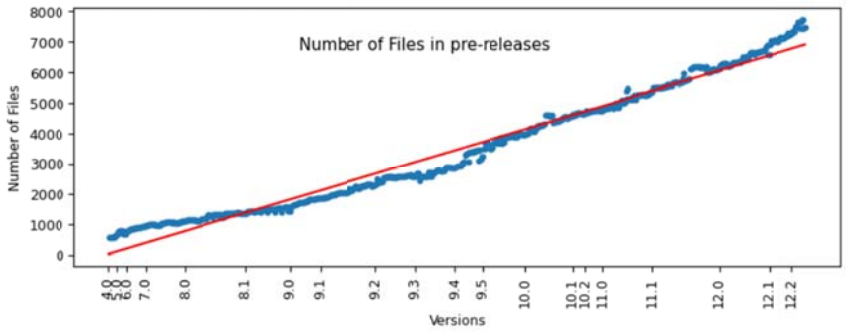

Fig. 9. Number of Files in Pre-releases

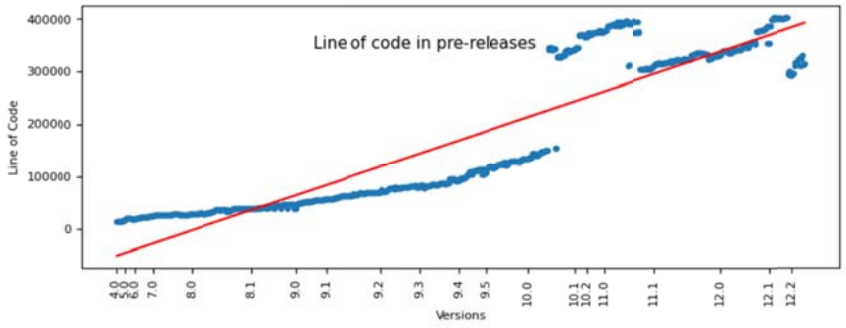

Fig. 10. Number of line of codes in Pre-releases

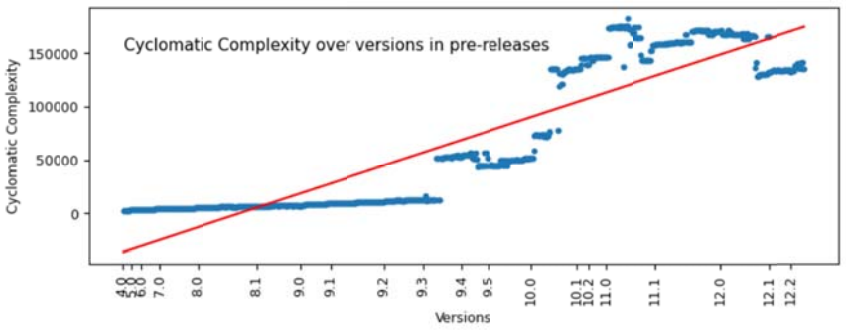

Fig. 11. Cyclomatic Complexity Over versions in Pre-releases

From figure 9, where we have fitted a linear regression line with number of functions in pre-release, we can see that the coefficient value of number of files 6.6 which is positive. So we can see continuing growth in the prereleases. Likewise, with the coefficient value of 201.87 and 424.32, we can see positive growth here. So by analysing the pre-releases, we can see that, it shows continuing growth.

E. Evolution of Angular

In this section, we investigate Lehman’s rules of software evolution using data that corresponds to the development of the Angular framework.

• Law 1: Continuing Change

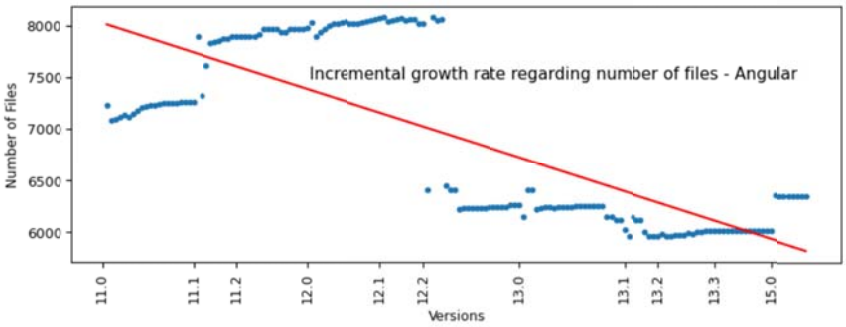

Fig. 12. Incremental growth rate regarding number of files – Angular

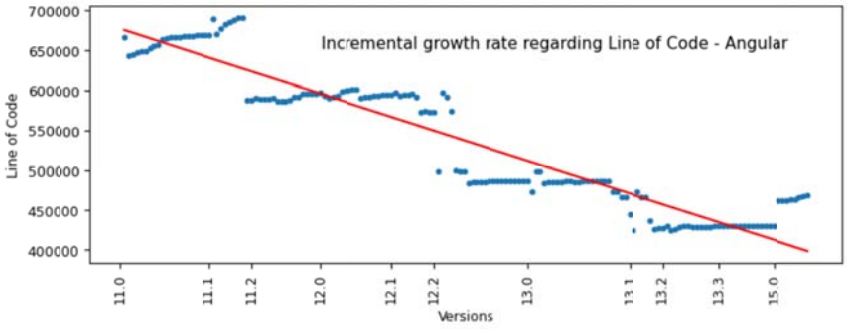

Fig. 13. Incremental growth rate regarding Line of Code - Angular

The graph in Figure 12 illustrates how the number of files increases over time. On the X-axis, we plotted the versions from 11.0 to 15.0, and on the Y-axis, we plotted the number of files. To comprehend the variations in the amount of files across versions, a regression line was fit. As of 2021, open source versions of Angular will be available, resulting in numerous releases throughout time. We determined the coefficient to be -13.60 while the intercept was 8022.42. Since the coefficient value is negative, we can conclude that Angular's version decline is linear. To do additional study, we plotted lines of code over versions and discovered a linear reduction with coefficient value -1722.16 and intercept value 677569.27. (Fig. 13) We can therefore conclude that the following program is incapable of showing continuous evolution and adaptation to its environment.

• Law 2: Self-regulation

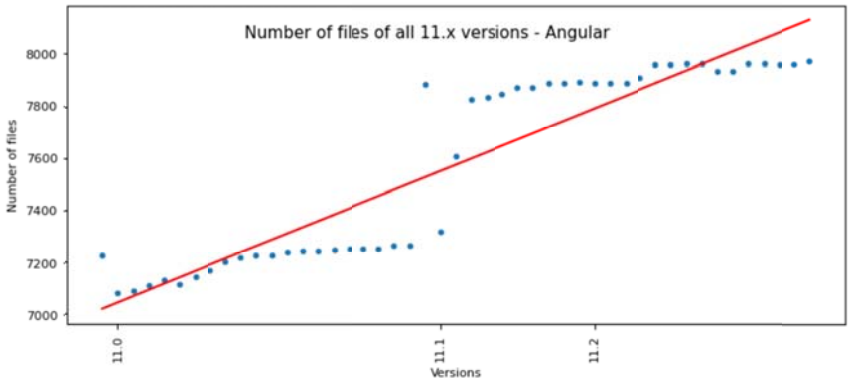

Fig. 14. Number of files of all 11.x versions - Angular

The graph in Figure 14 demonstrates how the number of files in stable releases of all eleven versions grows regularly. On the X-axis, we plotted the version number, and on the Y-axis, we plotted the number of files for all 11 versions. We used linear regression to identify the connection. The coefficient value is 24.18, and the intercept is 6994.91, indicating that there is a positive relationship between the number of files and all 11.X versions. Generally, observable development patterns are very constant, however there are minuscule changes that can be described as ripples. Occasionally, however, they exhibit larger increases due to the installation of an externally designed extra subsystem. Self-regulation or a consistent labor rate may be responsible for the growth's relative steadiness. The result is qualitatively similar to Lehman's findings, namely that the growth rate appears to oscillate around a mean and that particularly rapid growth is nearly always followed by sub-average growth, indicating an alternation between growth and stability. As a result, our findings can be interpreted as incidental evidence for the existence of self-regulation.

• Law 3: Conservation of organizational stability (invariant work rate)

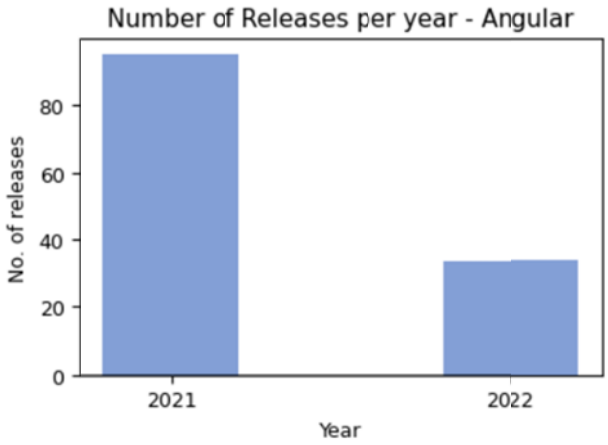

Fig. 15. Number of Releases per year - Angular

According to this law, the average effective overall rate of activity on a developing system will remain the same over the duration of the product's lifecycle. The total number of annual releases for the years 2021 and 2022 is shown in Figure 14. The year 2021 is likely to see the largest amount of releases.

• Law 4: Conservation of familiarity

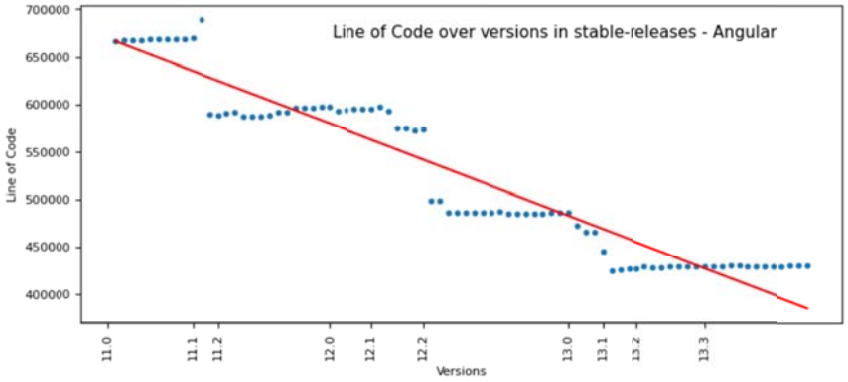

Fig. 16. Line of Code over versions in stable-releases – Angular

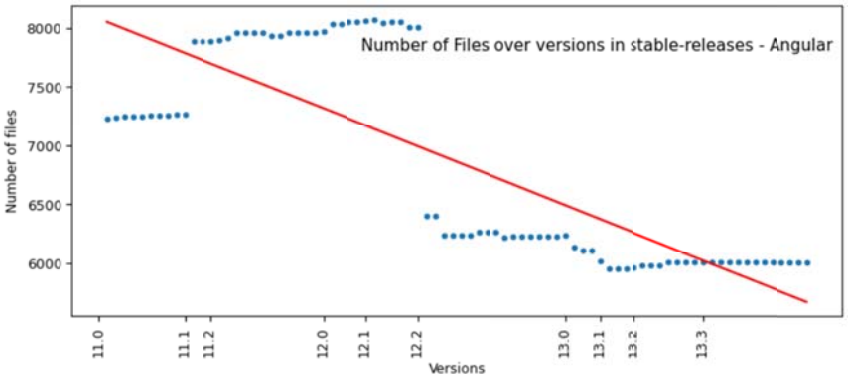

Fig. 17. Number of Files over versions in stable-releases – Angular

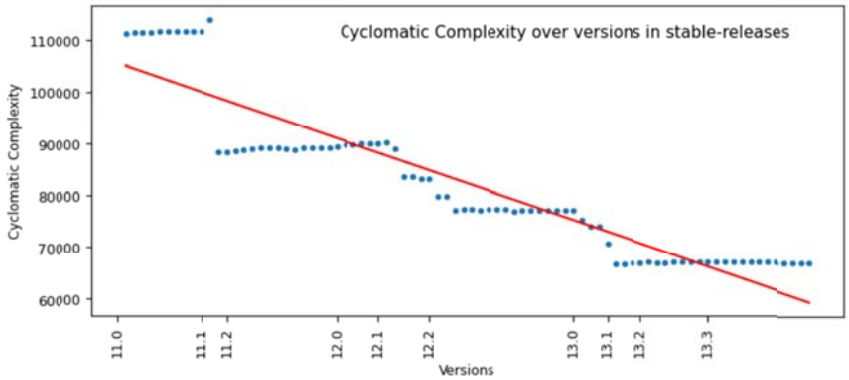

Fig. 18. Cyclomatic Complexity over versions in stable-releases

This law limits changes between releases so developers and users can preserve their code and system knowledge. Next.js preserves the familiarity when we analyze the stable versions and evaluate the lines of code, cyclomatic complexity, and number of files. The coefficient value for the number of files in stable versions is -29.44, which is negative and indicates a negative association with the number of files in ad versions. The coefficient values for lines of code and cyclomatic complexity were -3466.62 and -558.69. As both are negative, we can conclude that Next.js complies to Lehman's fifth law.

• Law 5: Continuing growth

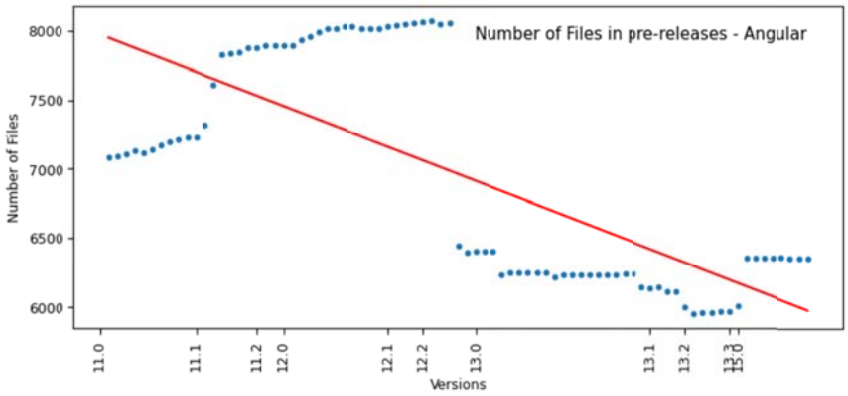

Fig. 19. Number of Files in pre-releases – Angular

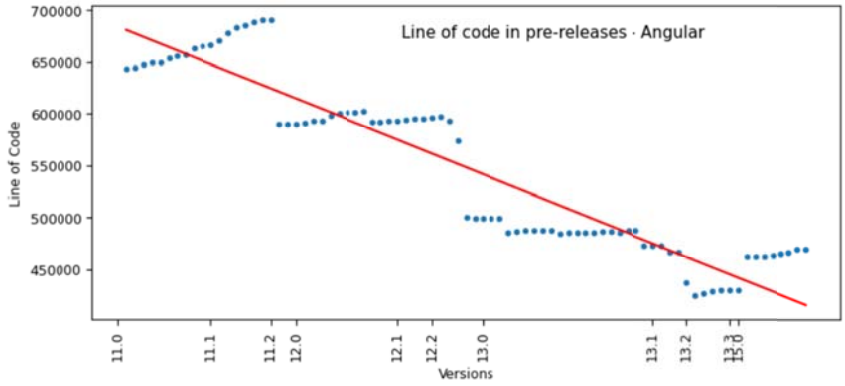

Fig. 20. Line of code in pre-releases – Angular

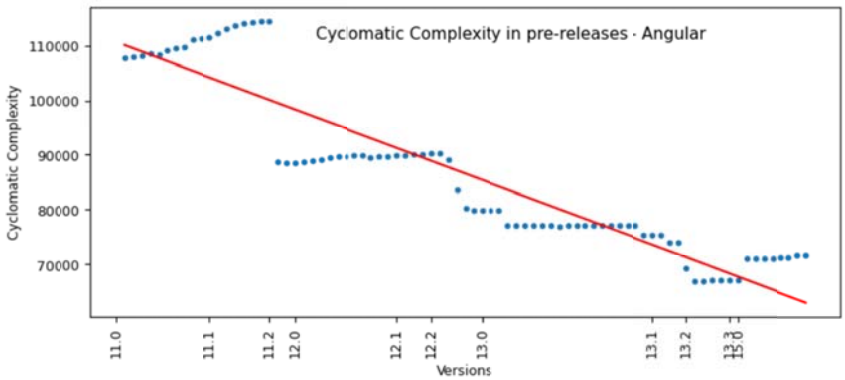

Fig. 21. Cyclomatic Complexity in pre-releases – Angular

The progression of the growth in the amount of files included in pre-releases is shown graphically in Figure 19. We plotted the versions along the X-axis, and the number of files that were included in pre-release builds was shown along the Y-axis. We fitted a regression line with the number of files and found coefficient value of -24.68 which shows a negative regression line. This means that Angular does not conserve the sixth law of continuing growth. BY analysing the pre-releases, we can know that the number of files is not increasing as the version is improving. But there can be a reason that, if the codes are being re-engineered, there may be fewer files in the following versions. But as Lehman said, we cannot say that there is continuous growth in that software if the number of files is not increasing.

When we investigate older versions of the software and examine the line of code, cyclomatic complexity, and number of files, we find that Angular does not maintain a level of familiarity that is consistent across all of these metrics. For Line of Code and Cycloamtic complexity Angular also showed negative regression line with coefficient value of -558.19 and -3328.01 respectively.

From this case study we can conclude that, Next.js followed the six laws of Lehman we have analysed. But to our surprise, we found out that, Angular did not follow the rules. Hence it showed total opposite characteristics. From the case study our findings say that, E-type software systems tend to grow with time (next.js). The developers and software designers need to keep in mind that with ever growing files and codes in the software, maintainability of the software becomes an issue. So to keep up with the continuous evolution of E-type software, dedicated teams should have budget for maintaining the software systems in the long run.

5. Conclusion

In the field of software engineering, software evolution has been an important part. Many software systems are being depricated for not keeping the pace with the real world. So to maintain the popularity of software, there is a vast need for maintaining the software systems as per the feedback of the developers and the testers.

Next.js and Angular are two of the most popular open source frameworks worldwide. In order to meet the needs of their users, they have continued to evolve throughout the past few years. In this paper, we tried to measure how these types of open source E-type softwares maintain the Lehman's laws of software evolution. This paper found out that, Next.js followed the laws. From the graphs and plots it is clear that, Next.js supported ongoing growth and continual change to support the laws.

Surprisingly, Angular being one of the most popular front-end frameworks, did not follow the laws that Lehman proposed. It showed total opposite behaviour in the graphs. It did not show probable support for a constant work rate. When new production versions are introduced, it appears that preservation of familiarity is coupled with significant modifications. The tool that has been used to analyse the software systems are not error-free. So there can be errors in analysing the software source-codes. Also, some version of codes is not available.

So there was some slight gap in the data that have been fetched from github. As we made a surprising finding of Angular not following the Lehman's laws, in the future there can be further research with this front-end framework to analyse why this is not following the laws. Also working with some other popular front-end frameworks can also be of a great path for further research.

This study works with two E-type software systems. The findings of the study suggest that, E-type software systems follow Lehman's Laws of software evolution. Also in software maintenance the research contributes greatly by giving an indication of the trends E-type software system follows. This work follows the similar methodology of related works and is of great use for the software developers, designers and the team.

References

[1] S. Bhatti, A. Dewani, S. Maqbool, and M. A. Memon, “A web based approach for teaching and learning programming concepts at middle school level.” International Journal of Modern Education & Computer Science, vol. 11, no. 4, 2019.

[2] Israeli, A. and Feitelson, D.G., 2010. The Linux kernel as a case study in software evolution. *Journal of Systems and Software*, *83*(3), pp.485-501.

[3] I. C. Nwandu, J. N. Odii, E. C. Nwokorie, and S. A. Okolie, “Evaluation of software quality in test-driven development: A perspective of measurement and metrics.”

[4] M. M. Lehman, “The past, present, and future of software evolution, in: Proceedings of frontiers of software maintenance,” European Workshop on Software Process Technology, pp. 108– 124, 1996.

[5] M. M. Lehman, "Programs, life cycles, and laws of software evolution," in Proceedings of the IEEE, vol. 68, no. 9, pp. 1060-1076, Sept. 1980, doi: 10.1109/PROC.1980.11805.

[6] “Programs, life cycles, and laws of software evolution,” Proceedings of the IEEE, 1980.

[7] M. M. Lehman, “Laws of software evolution revisited,” European Workshop on Software Process Technology. , Berlin, Heidelberg, 1996.

[8] Lehman, M.M. (1978). Programs, Cities, Students— Limits to Growth?. In: Gries, D. (eds) Programming Methodology. Texts and Monographs in Computer Science. Springer, New York, NY.

[9] D. Li, B. Guo, Y. Shen, J. Li, and Y. Huang, “The evolution of open-source mobile applications: An empirical study,” Journal of software: Evolution and process, vol. 29, no. 7, p. e1855, 2017.

[10] J. Zhang, S. Sagar, and E. Shihab, “The evolution of mobile apps: An exploratory study,” in Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile, 2013, pp. 1–8.

[11] M. I. Lone and Z. A. Wani, “Analysis of operating systems and browsers: A usage metrics.” Trends in Information Management, vol. 7, no. 2, 2011.

[12] M. W. Godfrey and D. M. German, “On the evolution of lehman’s laws,” Journal of Software: Evolution and Process, vol. 26, no. 7, pp. 613–619, 2014.

[13] Newhook, R., Jaramillo, D., Temple, J.G. and Duke, K.J., 2015. Evolution of the mobile enterprise app: A design perspective. *Procedia Manufacturing*, *3*, pp.2026-2033.

[14] Gonzalez‐Barahona, J.M., Robles, G., Herraiz, I. and Ortega, F., 2014. Studying the laws of software evolution in a long‐lived FLOSS project. *Journal of Software: Evolution and Process*, *26*(7), pp.589-612.

[15] Gupta, A., Suri, B., Kumar, V. and Jain, P., 2021. Extracting rules for vulnerabilities detection with static metrics using machine learning. *International Journal of System Assurance Engineering and Management*, *12*, pp.65-76.

[16] Kim, D.K., 2017. Finding bad code smells with neural network models. *International Journal of Electrical and Computer Engineering*, *7*(6), p.3613.

[17] “Angular versioning and releases,” https://angular.io/guide/releases, accessed: 2010-09-30.

Authors’ Profiles

Anika Tabassum has completed her B.Sc. (Hons.) in Software Engineering from Institute of Information Technology, University of Dhaka. Currently, she is conducting her Masters in Software Engineering in the same institute. She has interest in Federated Learning, Machine Learning and Software Maintenance and Evolution.

Ishrat Jahan Emu received the BSSE (Bachelor of Science in Software Engineering) degree and is currently pursuing the MSSE (Master's of Science in Software Engineering) in Institute of Information Technology, University of Dhaka. She is interested in working in the fields of bioinformatics, software engineering and machine learning.

Abdus Satter is an Assistant Professor at the Institute of Information Technology (IIT), University of Dhaka, Bangladesh. He pursued his Master of Science in Software Engineering (MSSE) and Bachelor of Science in Software Engineering (BSSE) from the same institution with the top score in his class. His core areas of interest are software repository mining, software engineering, web technologies, systems and security. He has numerous awards in various national and international software and programming competitions, and hackathons project showcasing.

How to cite this paper: Anika Tabassum, Ishrat Jahan Emu, Abdus Satter, "Software Evolution of Next.js and Angular", International Journal of Engineering and Manufacturing (IJEM), Vol.13, No.4, pp. 20-33, 2023. DOI:10.5815/ijem.2023.04.03