Software Reuse and Evolution in JavaScript Applications

Anastasia Terzi
Department of Electrical and Computer Engineering
University of Western Macedonia
Kozani, Greece
a.terzi@uowm.gr

Orfeas Christou
Department of Electrical and Computer Engineering
University of Western Macedonia
Kozani, Greece
ece01253@uowm.gr

Stamatia Bibi
Department of Electrical and Computer Engineering
University of Western Macedonia
Kozani, Greece
sbibi@uowm.gr

Pantelis Angelidis
Department of Electrical and Computer Engineering
University of Western Macedonia
Kozani, Greece
paggelidis@uowm.gr

***Abstract*—JavaScript (JS) is one of the most popular programming languages on GitHub. Most JavaScript applications are reusing third-party components to acquire various functionalities. Despite the benefits offered by software reuse there are still challenges, during the evolution of JavaScript applications, related to the management and maintenance of the third-party dependencies. Our key objective is to explore the evolution of library dependencies constraints in the context of JavaScript applications in terms of (a) the changeability (i.e., number of removed, added, or maintained libraries) (b) the update frequency of the library dependencies. For this purpose, we conducted a case study on the 86 most forked JavaScript applications hosted on GitHub and analyzed reuse data from a total of 2.363 successive releases. In general, 39% of the packages introduced in the first version of the project are being reused in the entire project's lifetime. The number of package dependencies slightly grows over time, while several other are being permanently removed. Regarding the evolution of third-party applications, it is observed that developers do not update the dependencies constraints to a most recent version, waiting to reach probably "breaking points" when the updates will be inevitable.**

Keywords— software reuse, JavaScript, software evolution, maintenance, changeability.

I. Introduction

Today, almost every computing device in the world, including desktop and mobile devices, sensors, and smart devices, have active JavaScript (JS) interpreters installed. The reason of JS popularity involves the sole characteristics of the language that provide interactivity by supporting the run-time, on-demand response to the end-users needs. The JavaScript developer community is a very active one and maintains over 1M packages on the npm[1] registry and more than 645K repositories on GitHub[2], that are ready to be reused and freely shared. JS developers nowadays have access to a plethora of development frameworks and libraries that are systematically maintained and can be reused to boost productivity and accelerate the development pace. A large number of reusable packages available makes program creation and evolution easier but does not come without challenges. Among these challenges is the need to maintain and update the reused third-party packages [2], [3], [16]. JS developers that reuse third-party libraries, often have to answer the following questions during the lifecycle of the applications:

- **Are existing third-party library dependencies sufficient to fulfil the application requirements at the time?** Often during the lifespan of an application, it is observed that reused libraries do not cover anymore the needs of the hosting applications. This may be caused either by the new functionalities or technologies adopted in the recent versions of the hosting application that are no longer compatible with the reused libraries or by the fact that the reused libraries are no longer maintained and therefore related bugs, fixes, and updates are not supported. In that case, developers are forced to replace packages with other related ones that offer the required functionality. This procedure is related to the changeability of the reused packages.
- **Should third-party library dependencies constraints be updated to a more recent version?** In this case, the developer should decide whether it is necessary to update a specific library constraint towards a newer version of the same library. Such a decision is related to the features added on the recent version, the compatibility compared to the current version, its popularity, and the community support [7]. A library dependency update in the hosting application may be optional (i.e., the update is not necessary for the hosting application to continue to operate) or mandatory (i.e. the hosting application will not be able to operate). In both cases, the process of updating a third-party dependency may require a lot of effort.

In this paper, our goal is to investigate **(a) the changeability of third-party library dependencies in JavaScript applications**. With the term changeability we refer to the libraries that are added, removed, or maintained in the hosting application during its evolution. We believe that such information will help JS developers understand whether the initial reuse choices were opportunistic [16] (i.e., indicative of reuse choices that during the project's lifespan need to be reconsidered) and should further on follow a more systematic approach to software reuse and **(b)**

[1] https://www.npmjs.com/

[2] https://github.com/

the update frequency of third-party library dependencies in JavaScript applications. Our goal is to explore to which extent third-party library dependency updates are performed in JavaScript applications. Such information will help JS developers organise and schedule future third-party library updates based on current evidence and practice. Specifically, we performed an embedded multiple case study on the 86 most forked JavaScript applications hosted on GitHub and analysed reuse data from 2.363 successive releases.

The rest of the paper is organised as follows: Section II discusses a summary of related research. Section III presents the case study design while Section IV presents the results organised by the research question, in Section V there is an interpretation of the results along with the threats to the validity of our study. In Section VI, we conclude the paper.

II. Related Work

The evolutionary process for applications that facilitate third-party reuse is quite different from applications built completely from scratch since software developers need to update both the part of the application implemented internally and the dependencies to third-party libraries [7]. Currently, in literature, some studies examine third-party library reuse evolution [4], [7], [14], [15], [19] from the scope of packages hosted on npm. Zerouali et al. [20] examined the update lag of library dependencies in package networks, to assess how outdated a software package is compared to the latest available releases of its dependencies. The authors observed that developers in order to avoid backward-incompatible changes are using strict dependency constraints or the exact version number leading to technical lag. In their research they pointed out that even though npm packages are constantly updating the dependency constraints on the same packages are not being updated, increasing the likelihood of dependent packages suffering from an increased technical lag. The researchers concluded that developers are more likely to update dependencies in major version project upgrades than in minor or micro-updates while they suggested developers not start using newly available packages immediately because of the possible bugs.

In general, third-party dependencies are not updated on a regular basis, while the reuse evolution depends on the project evolution rate according to Kula et al. [8]. For projects that present a high evolution rate (indicated by frequent releases) third-party dependencies are added more regularly, and the changes are of a lower scale, while for projects that present a low evolution rate third-party library dependencies evolve less frequently but with a greater influence on the system [8]. The difference rate between a package update and a project dependencies update causes the technical lag problems that will increase over time, even in the beginning of a project's lifetime [5]. The problem increases even more as developers prefer to even downgrade library dependencies for the sake of project stability [8], [20]. Stringer et al. [15] studied the update lag of third-party dependencies and concluded that this lag is slightly correlated with the amount of change introduced in a new version of the reused library, in the sense that many changes to the new library version tend to increase the update lag. Also, this study highlights that the update of a library depends on the importance of the library for the project [15]. This study experiments in JavaScript applications along with Java and concludes that even though the majority of dependencies are outdated the lag is identified in 1 or 2 versions before [15]. Despite the frequency of the project update, studies have shown [8], [15], [19] that developers response to a library update opportunity is slow and lagging. This finding is also confirmed by Zaimi et al. [19] that concluded that once a library is imported into a Java system, it is unlikely to be deleted or changed to a more recent version [19]. The same research reveals that when library deletions and updates occur, the most likely reason is a re-assessment of a reuse decision in the previous version, i.e., the addition of multiple libraries that did not fit well into the project, and not necessarily the upgrade of the system itself [19]

According to Seo et al. [14] even well-thought modifications such as the removal, addition, or upgrade of a library may result in system problems and quality degradation, which may lead to a system crash [14]. As a response, developers are cautious, and changes to third-party library dependencies are implemented slowly or not at all. This method of handling third-party dependencies causes the common build issue [14]. Cox et al. [4] suggested that the selection of the most appropriate version of a dependency can be a) context-specific; b) the most stable version, c) a long-term support version, or d) the latest version of the library dependency [4].

As part of our research, we are going to quantify the evolution of dependencies in the context of applications developed in JavaScript programming language, to further explore previous findings.

III. Case Study Design

The goal of the research is to examine the evolution of third-party library dependencies in the context of applications developed in JavaScript programming language. For this reason, we conducted a case study and analysed 20 versions of 86 JavaScript projects hosted on GitHub. In this section, we describe the case study, which was designed and reported according to the guidelines proposed by Runeson and Host [12]

A. Goal and Research Questions

The goal of this study, described with the help of the Goal-Question-Metric formalism is: "to analyse third-party library dependencies with respect to (a) the changeability of the library dependencies and (b) the release updates of library dependencies constraints on the point of view of software engineers in the context of JavaScript application development". Therefore, we formulated the following Research Questions (RQs):

[RQ1]: Is there a trend in the changes observed in the library dependencies constraints of JavaScript applications? In this question, we want to examine the number of library dependencies that are added, removed, or maintained in consecutive releases of the hosting application during its evolution. Such information will help JS developers understand whether initial reuse choices were well-aimed (i.e., indicated by dependencies during the project's lifespan that remains stable) or whether these choices were often reconsidered.

[RQ2]: Is there a trend in the update frequency of library dependencies in JavaScript applications? Our goal here is to explore the extent to which third-party library dependency updates are performed in JavaScript applications. Outdated library dependencies may involve potential risks related to API incompatibility, security threats, bug fixes etc. Therefore, the answer to this question will help JS developers organise and schedule future third-party library updates based on current evidence and practice.

B. Selection of cases

The case study of this paper is a multiple-case study [12], the context is open-source JavaScript applications, and the cases of analysis are the third-party library dependencies observed in the projects. Overall, we gathered data from 86 JS applications hosted on GitHub and analysed 2.363 releases presenting 98.638 library dependencies.

The criteria that we have used for selecting projects are discussed below:

- JavaScript should be the major scripting language in which the projects are developed.
- Projects should have a lifespan of at least 2 years
- Projects should have at least 1k stars on GitHub
- Projects should present at least 20 releases to justify evolution analysis (this information is provided by the GitHub repositories)

The process we followed to retrieve reuse information from JS applications hosted on GitHub is the following: Initially, we browsed JavaScript projects and sorted the results by the most to least forked. Forks are frequently used in open-source software to test ideas or improvements before submitting them back to the main repository and can be used as a metric factor to indicate both the popularity and the sustainability of a project [13]. After getting all the results, we filtered out the ones that did not match the aforementioned criteria. 100 projects have been initially chosen, out of which 14 were excluded due to a lack of publicly available dependency data. To clarify the versioning factor, we should mention that we did not rely on Semver since there is a large number of developers that do not follow the suggested way of package, and project versioning [18] and we did not want to exclude any successive project's version. For each project, the library dependencies were recorded based on configuration files (package.json and package-lock.json files). These files support various mechanisms (e.g., inheritance, version range, variable expansion, and dependencies trees) to declare library dependencies. We developed a tool that extracts a library dependency via parsing three fields: Id, type of dependency and dependency constrain version for each release. For each of the library dependency, we retrieve from npm, data about the date of release and the total number of these library versions. These data were used to calculate the metrics presented in Table I to identify trends related to the changes and the updates of the library dependencies observed in JS applications. The same data were used to calculate the trend in the evolutionary behaviour of the project.

C. Data Analysis

In order to explore the research questions, we performed descriptive statistical analysis and hypothesis testing. The analysis plan is presented in Table II. To answer the RQs of the study we followed a similar process where we employed:

- **descriptive statistics** (i.e., min, max, median, and standard deviation) to examine the related metrics, for each JS application under study separately. Also, for several of these metrics ([V12], [V13], [V14], [V15], [V16], [V17]) we calculated the accumulative descriptive statistics for all participating JS applications.
- **box-plots** to visualise the distribution of values of certain metrics ([V3], [V4], [V5], [V6], [V7], [V8]) for all participating applications. Also, we adopted line charts to visualise the evolution of the number of library dependencies ([V3]), across all project's version, for every application separately.

I. DEPENDENCY METRICS

Alias	*Metric*
$[V_1]$	Number of library dependencies that exist in a project version
$[V_2]$	Number of removed library dependencies on each project version
$[V_3]$	Number of added library dependencies on each project version
$[V_4]$	Number of removed library dependencies that are re-added on next versions
$[V_5]$	Total number of dependency changes in each project version
$[V_6]$	Number of updated library dependencies that exist in every project version
$[V_7]$	Number of outdated library dependencies that exist in every project version
$[V_8]$	Number of library dependencies that exist in every project version
$[V_9]$	Total number of library dependencies observed in the project
$[V_{10}]$	Total number of removed library dependencies
$[V_{11}]$	Total number of added library dependencies
$[V_{12}]$	$\frac{[V_8]}{[V_9]}100\,\%$ Percentage of library dependencies that exist in all project versions
$[V_{13}]$	$\frac{[V_4]}{[V_{10}]}100\,\%$ Percentage of non permanent library dependencies removal
$[V_{14}]$	$\frac{[V_3]}{[V_{11}]}100\,\%$ Percentage of added dependencies during project's lifetime
$[V_{15}]$	$\frac{[V_{10}]}{[V_7]}100\,\%$ Percentage of removed dependencies during project's lifetime
$[V_{16}]$	$\frac{[V_6]}{[V_8]}100\,\%$ Percentage of updated library dependencies
$[V_{17}]$	$\frac{[V_7]}{[V_8]}100\,\%$ Percentage of outdated library dependencies

- the **Man-Kendall trend test** for hypothesis testing. The Man- Kendall trend test [10] involves the following hypotheses:

  H_0: There is no trend supported by the software data analysed, so the RQ cannot either be confirmed or contradicted.

H1: There is a negative, non- null, or positive trend regarding the RQ.

n this case we formed pairs that include the initial version ınd the latest version of the application under study, where ve explored whether there is a trend in the evolution of the ibrary dependencies. For RQ1 we examined whether ⁄ariables [V2], [V3], and [V4] present a particular trend increasing, decreasing or remain stable through each project's lifecycle). For RQ2 we examined [V6], [V7], V16] and [V17] variables.

I. DEPENDENCY METRICS

Research Question	*Metrics*	*Analysis*
[RQ1]	$[V_{12}], [V_{13}], [V_{14}], [V_{15}]$ $[V_1], [V_2], [V_3]$ $[V_8], [V_{10}], [V_{11}]$ $[V_3]$	Descriptive Statistics Mann-Kendall Analysis Box-plots Line chart
[RQ2]	$[V_{16}], [V_{17}]$ $[V_6], [V_7]$ $[V_6], [V_7]$	Descriptive Statistics Mann-Kendall Analysis Box-plots

V. RESULTS

n this Section we present the results of this case study, organised by research question.

RQ1]: Is there a trend in the changes observed in the ibrary dependencies of JavaScript applications?

II. DESCRIPTIVE STATISTICS-RQ1

Variable	*N*	*Minimum*	*Maximum*	*Median*	*Std.Deviation*
$[V_{12}]$	86	0%	100%	36.29%	35.0%
$[V_{13}]$	86	0%	97,01%	6,78%	19,81%
$[V_{14}]$	86	0%	668%	42%	75%
$[V_{15}]$	86	0%	700%	27%	77%

n this question we examine whether the number of library dependencies that are added, removed or maintained in consecutive releases of the hosting JavaScript applications present a particular trend. To answer RQ1 we:

- Extracted the dependencies on every project version of the JS applications under study based on the information of package.json and package-lock.json files. Then we calculated the number of dependencies that exist in every project version [V8] and the total number of dependencies [V9] used accumulatively by each project.
- Analysed the dependencies on each version of the project to determine which were imported [V3] and which were removed [V2] in subsequent versions.
- Produced a list of dependent libraries that remained consistent throughout all versions [V12].
- Calculated the values of percentage variables [V14] and [V15].

Table III presents the descriptive statistics of percentage ⁄ariables ([V12], [V13], [V14], [V15]) for all projects while in Table IV we present the values of variables [V4], [V8], [V9] and [V10] for every project. Figure 1 visualises with the help of box-plots the number of dependencies that are (a) added [V10], (b) removed [V11], and (c) maintained on project level [V8]. As shown in Table IV less than 50% of the total amount of third-party dependencies are maintained in all project versions. As shown on Figure 1 projects present the same evolution trend. Although there are projects (i.e., browserify) where we observe sudden increases/ decreases in the number of the dependencies (see [V11], Table VI) that can be interpreted as wrong reuse choices that are subsequently corrected. The number of third-party dependencies presents overall small increases and decreases through the evolution of projects as shown in Figure 2.

IV. RESULTS OF ALL METRICS ON ALL PROJECTS

Project	*V4*	*V6*	*V7*	*V8*	*V9*	*V10*	*V11*	*Project*	*V4*	*V6*	*V7*	*V8*	*V9*	*V10*	*V11*
amazeui	13	10	1	11	54	21	32	lazysizes	0	0	1	16	16	0	6
anime	0	0	5	5	6	6	6	leetcode	0	1	1	1	2	0	1
appwrite	6	0	1	1	13	8	6	material	0	58	0	22	80	2	7
async	0	18	18	36	39	1	4	material-ui	81	187	29	189	376	85	180
ava	1	41	6	47	133	39	22	medium-editor	0	20	5	25	25	0	4
awx	0	0	20	20	52	2	12	MQTT.js	1	39	6	7	46	9	7
axios	0	24	4	28	44	4	8	mustache.js	0	10	0	0	10	0	3
bootstrap	0	10	0	10	111	37	33	node-soap	0	45	2	3	48	5	19
browserify	65	0	34	34	66	67	441	noVNC	0	51	0	8	59	26	49
card	0	1	0	1	38	16	17	openlayers	2	81	4	7	88	23	39
cesium	0	0	15	15	82	2	34	p5.js	0	87	0	0	87	32	31
ChakraCore	0	0	5	5	5	0	0	paper.js	0	34	8	9	43	4	8
Chart.js	0	20	33	53	55	0	4	pdf.js	0	98	0	0	98	24	42
clipboard.js	1	1	0	1	46	15	22	pdfmake	0	37	4	7	44	10	18
core-js	0	37	18	55	79	12	47	perfect-scrollbar	0	23	0	1	24	19	17
cropperjs	0	12	0	12	50	6	30	pixijs	0	114	0	0	114	17	30
cytoscape.js	0	0	14	14	34	4	4	plotly.js	0	33	66	104	137	10	10
dash.js	0	0	6	6	80	32	31	pouchdb	44	84	2	3	87	76	49
discord.js	0	1	0	1	43	17	32	prettier	0	74	11	84	158	26	36
dropzone	0	4	0	4	39	12	27	progressbar.js	0	28	4	4	32	1	18
EaselJS	0	0	0	0	15	2	7	protobuf.js	0	69	0	1	70	19	40
editor.md	0	1	8	9	19	6	5	ScrollMagic	0	36	0	1	37	11	28
element	1	4	34	38	82	2	1	sequelize	0	68	0	0	68	8	12
ember.js	0	33	62	95	103	3	4	showdown	0	5	3	17	22	2	2
eslint	0	35	45	80	88	4	7	stackedit	1	16	38	85	101	8	5
express	0	24	1	25	53	7	24	summernote	0	108	0	1	109	6	13
fetch	0	1	0	1	21	1	18	svgedit	0	25	7	30	55	23	17
fine-uploader	0	9	2	11	13	3	1	svgo	0	52	0	0	52	2	17
fingerprintjs	0	0	0	0	55	7	23	sweetalert2	1	14	14	18	32	8	1
flot	1	0	1	1	1	7	0	typed.js	0	25	0	0	25	0	20
form	0	0	0	0	15	1	7	typeorm	0	73	70	70	73	0	0
Ghost	0	7	6	13	163	6	15	UglifyJS	0	0	2	2	2	0	0
grunt	1	1	10	11	25	2	0	velocity	0	29	0	1	30	0	7
hexo	0	1	3	4	48	14	12	video.js	0	34	44	45	79	1	7
highlight.js	0	12	5	17	35	2	20	vue-resource	1	31	0	3	34	9	14
howler.js	0	0	2	2	3	0	6	webogram	0	25	8	8	33	2	14
imagesloaded	0	1	12	13	17	4	3	webpack	0	13	92	92	105	1	0
intro.js	0	1	0	1	42	0	39	webtorrent	0	16	44	44	60	5	1
jasmine	0	1	0	1	26	5	6	wekan	0	11	27	27	38	6	4
johnny-five	0	1	10	11	24	2	1	ws	1	19	0	0	19	9	7
jsPDF	0	0	0	0	75	20	53	wtfjs	0	0	11	12	12	0	1
karma	0	0	21	21	78	13	3	x-spreadsheet	0	0	25	25	25	0	0
knex	0	10	5	15	56	6	16	zeroclipboard	1	29	0	0	29	19	17

We run Man-Kendall for variables: (a) [V1] (number of library dependencies in project version) we found that there is an increasing trend in the number of library dependencies through time, since 65 out of 86 projects presented significant p-value for sign.value<0.01. (b) [V2] (number of removed library dependencies in a project version) we did not find any significant trend and therefore the null hypothesis cannot be rejected and for (c) variable [V3] (number of added library dependencies in project version) where the null hypothesis cannot be rejected.

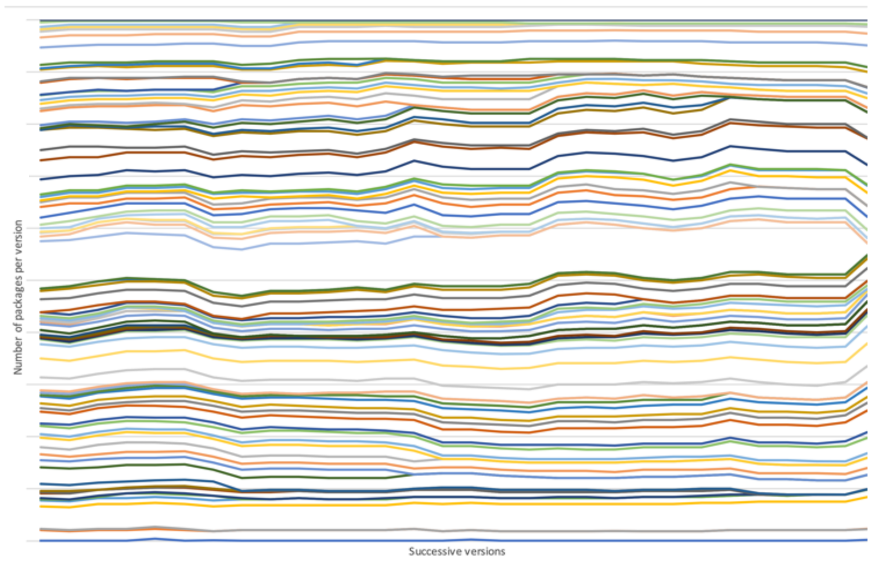

Fig. 2. Changes on the amount of dependencies on each project version

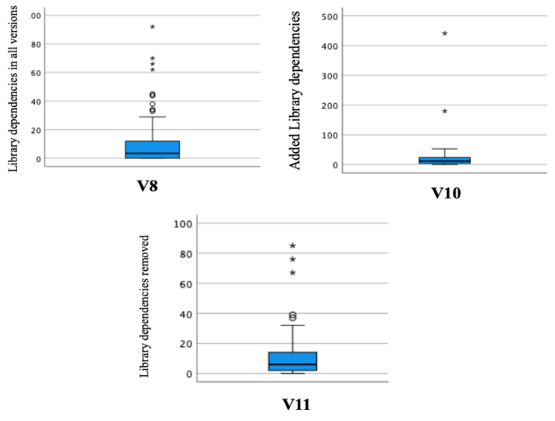

Fig. 1. Box-plots Changeability

[RQ₂]: Is there a trend in the update frequency of library dependencies in JavaScript applications?

Our goal here is to explore the extent to which third-party library dependency updates are performed in JavaScript applications. For this purpose, we examine the number of different versions of a library that are used during the course of the project.

V. DESCRIPTIVE STATISTICS-RQ₂

Variable	**N**	***Minimum***	***Maximum***	***Median***	**Std.*Deviation***
$[V_{16}]$	86	0%	100%	39%	41.99%
$[V_{17}]$	86	0%	100%	46%	43.17%

For the RQ2 we followed the procedure detailed above:

- For the library dependencies that are maintained in all versions of the project (as a number indicated in [V8], calculated on RQ1) we recorded whether updates were performed on the dependency constraints in each project version. In the end, we classified the library dependencies into two groups based on whether their constraints were updated or not and derived the following metrics:

[V6] represents the library dependencies that have been upgraded at least once in the project's lifetime and,

[V7] represents the library dependencies that have not been upgraded.

- Next, we calculated the values of percentage variables [V16] and [V17].

In Table IV we present the values of variables [V16] and [V17 for every project while Table V presents the descriptive statistics of percentage variables ([V16], [V17].) for all projects. Figure 3 visualises with the help of box-plots the number of library dependencies that are (a) outdated ([V7]) and (b) updated ([V6]) at the project level. As we can see in Table V on average a JavaScript application updates reach 39% of the libraries, while 46 % of the libraries remain outdated, the rest of 15% are the libraries that are libraries that do not employ a specific version (i.e., in the package.json file this is indicated by a *). From the box-plots of Figure 3, we see that most projects present the same update frequencies while several projects performregular updates (i.e., webpack, grunt, browserify, eslint.js) and projects that do perform very scarcely updates (i.e., bootstrap, express).

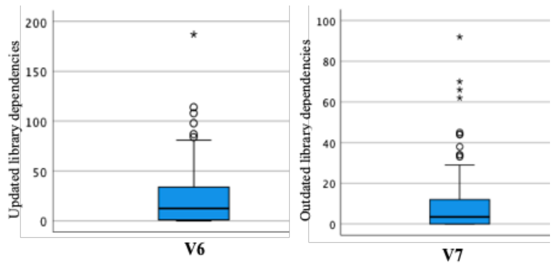

Fig. 3. Box-plots Update of library dependencies

For the RQ we run Man-Kendall for variables: (a) [V7] (number of outdated library dependencies) were we found that there is an increasing trend in the number of library dependencies that are outdated through time since 72 out of

86 projects presented significant p-value <0.01. (b) [V6] (number of library dependencies that are updated where we did not find any significant trend and therefore the null hypothesis cannot be rejected.

V. Discussion

In this section, we first interpret the results of the research questions and then we discuss the threats to validity of the research.

A. Interpretation of Results

This study examined the evolution of third-party library dependencies in the context of JavaScript application development. Most of the results are in accordance with existing literature, while there are some results that are surprising. We observed that:

The number of library dependencies remain relatively stable presenting a small increase during the evolution of JS applications. It seems that overall JS developers reuse a pre-defined number of libraries that implement specific functionality. New libraries may be added during the project's evolution, but this increase remains stable as we did not observe sudden peaks in the reuse intensity. This finding agrees with Zerouali et al. [20] who concluded that on package level the number of dependencies, rarely changed. They point out that dependencies are added or removed mostly in major releases, but they did not determine if there is a change at the total number of dependencies. Our result shows that developers overall preserve the organizational stability of the applications in terms of third-party dependencies keeping under control the maintenance effort required to manage the dependencies. Zaimi et al. [19] also argue that reuse intensity presents an increase over time. The result implies that even though developers tend to use more and more third-party packages in their projects there is a concern of increasingly adding features during project's lifetime.

The majority of reuse decisions are often revisited: Despite the fact that overall, the number of dependencies in JS applications remain stable, it seems that within successive versions there are changes (both additions and removals) in third-party dependencies. Usually, library removals occur simultaneously with library additions, a fact that implies a library substitution and not a removal of functionality. This finding is in contrast with Zaimi et. al [19] who argue that reuse decisions are not revisited in Java applicatios, while Zerouali et al. [20] showed that on package level dependencies are revisited at least on major updates of the package. In our case this finding can be explained by the fact that we chose to examine highly forked projects which means that third-party developers suggest changes to improve the initial project. Those changes are put under test in the main repository and sometimes are being removed on the next versions. Beside that JS developers in general seem to be more informed and experienced when reusing third-party dependencies in the sense that for several reused functionalities they are willing to test through the application's lifecycle several different emerging choices.

The library dependencies that remain during the evolution of the project are limited: Overall, just 39% of the initial dependencies remain unaltered in the applications until the latest versions. By carefully examining the libraries that remained stable we reached the conclusion that these are the libraries that are involved in the structural development of the application (i.e., programming or testing frameworks, compilers) and therefore are more difficult to replace compared to those that have not an important role in the application core (i.e., chart libraries, graphics).

Library version update is sparse: In this finding we reach an agreement with prior studies [6], [8], [19], [20] as we have verified that JS developers' response to library update opportunities are slow and lagging causing huge technical lag. This is a sign that JS developers hesitate to systematically update third-party dependencies probably due to the effort required, and the risk of introducing instability to the hosting application.

B. Threats to Validity

This section presents Runeson and Höst's [12] four key forms of threats to validity for quantitative research in software engineering: construct, internal, external, and reliability validity. ***Construct validity*** refers to how well an experiment performs in relation to its claims. In this study construct validity is subject to the selection of the metrics adopted to monitor the evolution of third-party libraries in JavaScript applications, that may not precisely reflect the phenomenon under study. To mitigate this threat, we selected metrics for quantifying the library dependencies evolution trend that (a) are already employed by related literature [8], [17], [19] (b) can be directly available from publicly available dependency managers (c) are calculated automatically with the help of tools, excluding vulnerabilities introduced by manual, subjective calculations. ***Internal validity***, in this case is not applicable since the examination of causal relationships is out of the scope of the study.

Regarding ***External validity***, that refers to the extent to which the results of a study are generalisable (i.e., represent the entire population) we identified two threats. The study findings are limited to third-party library dependencies in JavaScript applications and therefore cannot be generalised in applications developed in other Programming languages. Additionally, we used a rather limited sample of 86 JS applications, therefore we encourage the replication of the study in applications developed in different languages and in more samples. In order to increase the ***Reliability*** of the study, that reflects the reproducibility of a study, i.e. defined as the capacity of other researchers to duplicate the same process and reach the same conclusions we applied two mitigation actions: (a) we recorded the case study design protocol in detail and (b) we uploaded the relevant tools that were used to obtain the data, along with the collected data in a GitHub repository.

VI. Conclusions

The goal of this work was to investigate the evolution of library dependencies in the context of JavaScript applications in terms of (a) the changeability of library dependencies (i.e., the number of removed, added, or maintained libraries) and (b) the updates performed in the versions of the library dependencies. For this purpose, we performed a case study on the 86 most forked JavaScript applications hosted on GitHub for this purpose, and we examined reuse data from 2.363 subsequent releases.

The findings concerning the changeability of third-party library dependencies, demonstrate that in JS applications new library dependencies are frequently added and several libraries are simultaneously removed, while the total number of dependencies presents a slight increase over time. Also, we observed that 39% of the total number of library dependencies are maintained in all studied project versions without a change in the version constraints. These libraries usually represent the frameworks on which the core functionality of the hosting application is built. In most of the cases developers prefer to keep the library dependencies in outdated versions, probably in an attempt to lower the risk of incompatibilities that a new version may cause. As a future work we intend to work on methods that will support the developers in the process of updating dependencies in more recent versions. Specifically, we plan to work on methods for tracking the changes caused by the updates and the level to which third-party library interdependencies are affected.

Acknowledgment

This research was co-funded by the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship, and Innovation, grant number T1EDK-04873.

References

1. H. Borges, A. Hora, and M. T. Valente, "Understanding the Factors That Impact the Popularity of GitHub Repositories," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct. 2016, pp. 334–344.
2. E. Constantinou and I. Stamelos, "Architectural stability and evolution measurement for software reuse," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, Apr. 2015, pp. 1580–1585.
3. E. Constantinou and I. Stamelos, "Identifying evolution patterns: a metrics-based approach for external library reuse," *Software: Practice and Experience*, vol. 47, no. 7, pp. 1027–1039, Mar. 2017.
4. J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, "Measuring Dependency Freshness in Software Systems", *IEEE International Conference on Software Engineering*, May 2015, pp. 109–118.
5. A. Decan, T. Mens, and E. Constantinou, "On the Evolution of Technical Lag in the npm Package Dependency Network," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2018, pp. 404–414.
6. A. Javan Jafari, D. E. Costa, R. Abdalkareem, E. Shihab, and N. Tsantalis, "Dependency Smells in JavaScript Projects," in *IEEE Transactions on Software Engineering*, 2021, pp. 1–1.
7. K. Kaur, "Analyzing Growth Trends of Reusable Software Components," in *Designing, Engineering, and Analyzing Reliable and Efficient Software*, 2013, pp. 40–54
8. R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?," in *Empirical Software Engineering*, May 2017, vol. 23, no. 1, pp. 384–417.
9. W. C. Lim, "Effects of reuse on quality, productivity, and economics," *IEEE Software*, vol. 11, no. 5, pp. 23–30, Sep. 1994.
10. H. B. Mann, "Nonparametric Tests Against Trend," *Econometrica*, vol. 13, no. 3, p. 245, Jul. 1945.
11. P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz, "An empirical study of software reuse vs. defect-density and stability*26th International Conference on Software Engineering*, pp. 282–291
12. Per Runeson, HöstM., Austen Rainer, and Björn Regnell, *Case Study Research in Software Engineering Guidelines and Examples*. Hoboken, Nj, Usa John Wiley & Sons, Inc, 2012.
13. G. Robles and J. M. González-Barahona, "A Comprehensive Study of Software Forks: Dates, Reasons and Outcomes," in *IFIP Advances in Information and Communication Technology*, 2012, pp. 1–14.
14. H. Seo et al., "Programmers' build errors: a case study (at google)," in *Proceedings of the 36th International Conference on Software Engineering*, May 2014, pp. 724–734.
15. J. Stringer, A. Tahir, K. Blincoe, and J. Dietrich, "Technical Lag of Dependencies in Major Package Managers," in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, Dec. 2020, pp. 228–237.
16. A. Taivalsaari, T. Mikkonen, and N. Makitalo, "Programming the Tip of the Iceberg: Software Reuse in the 21st Century," in *45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug. 2019, pp. 108–112.
17. A. Terzi, S. Bibi, and P. Sarigiannidis, "Reuse Opportunities in JavaScript applications," *47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2021, pp. 387–391.
18. E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the JavaScript package ecosystem," *13th International Workshop on Mining Software Repositories - MSR '16*, 2016, pp. 351–361.
19. A. Zaimi *et al.*, "An Empirical Study on the Reuse of Third-Party Libraries in Open-Source Software Development," *7th Balkan Conference on Informatics Conference*, Sep. 2015, pp. 1–8.
20. A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, "An Empirical Analysis of Technical Lag in npm Package Dependencies," in *New Opportunities for Software Reuse*, 2018, pp. 95–110. doi: 10.1007/978-3-319-90421-4_6.