Vol.2018-SE-198 No.14
2018/3/9

SoL Mantra: Using Library Coexistence Coefficient to Visualize Update Opportunities.

Boris Todorov[1,a)] Raula G. Kula[2,b)] Takashi Ishio[2,c)] Katsuro Inoue[1,d)]

Abstract: In software development, software reuse has become a pivotal factor in creating and providing high-quality software at a reduced cost. The reuse of a code creates dependencies, which as they increase over time they become difficult to manage and could lead to compatibility issues or bugs if not kept up to date. With newer version releases, come various quality improvements, new features and issue fixes, but deciding whether or not to adopt those is a difficult task for new developers and for large software with a lot of dependencies.
To address these difficulties, we propose the SoL Mantra tool that visualizes update opportunities by applying coexistence coefficient between libraries in a software ecosystem based on which we display the information about the complexity of each update opportunity. The orbital layout provides the means to visualize the update opportunities and demonstrate its merits by showcasing examples from the JavaScript ecosystem. Through these examples, we further elaborate how maintainers can benefit from SoL Mantra's visual cues.

Keywords: Visualization, Library, Update

1. Introduction

For the past decade, the predominant practice within software engineering has been the usage of third-party software, also known as software libraries[1]. The core lies in the concept of software reuse, which reduces the man-hours cost when developing new software, inherited safety and stability from a code that is used by a vast majority of developers. But as most good things it has its drawbacks, as time passes, the libraries grow older and new versions are released, providing various benefits which include but are not limited to, new features, bug fixes, optimization.

The new versions also carry their own risks for the software developers that use them in their projects. Updating to a newer version could bring the before mentioned benefits, but can also cause devastating problems to the software[2],[3]. These include but are not limited to, potential compatibility issues with other libraries adopted by the software, requirements to restructure different sections of the code, due to method name alterations.

To use these libraries, the developers must add references within their code to the library itself, which creates a dependency to the said library[4]. Researchers have observed and document empirically how software developers interact and deal with library updating. Some even conclude based on a large sample of Java clients that use Maven libraries, that high number of systems keep their dependencies outdated [5]. Others provide in-depth analysis on the impact of newer version releases and what changes within them drive developers to adopt newer versions [6], [7].

Software developers, have to carefully evaluate the potential risks while making decisions "if" and "when" to update their software's library dependencies. There exist studies that have been conducted to express the concerns regarding incompatibility when updating [8], [9]. On the other hand, not updating a library could also lead to problems, such as the heartbleed bug[*1]. To keep those bugs from occurring the libraries must evolve over time, making additions to their functionalities and addressing various issues that the previous version might have. This evolution is represented by the releases and their corresponding versions supported by documentation.

This paper's goal is to create a tool that assists the software developers in recognizing potential risks[10],[11] while considering a library in their software for an update. To this end we adopted a relatively new binary concept called coexistence coefficient[12] and integrated together with the intricate orbital layout to create a interactive visual aid - the Software Library Mantra tool (SoL Mantra). The tool provides information whether or not a system dependency is outdated, by using simple and intuitive visual elements. Furthermore, to illustrate the potential risk from updating a library, we evaluate the popular library usage in the ecosystem and offer suggestions, which libraries should be updated together if either of them is considered for updating. The main technical challenges in achieving results lie in finding a balance between visualization technique and providing enough information without it being unreadable.

To evaluate the tools effectiveness we conducted an experiment consisting of 23 JavaScript based software systems acquired

1 Osaka University, Osaka, Japan
2 Nara Institute of Science and Technology, Nara, Japan
a) boris-t@ist.osaka-u.ac.jp
b) raula-k@is.naist.jp
c) ishio@is.naist.jp
d) inoue@ist.osaka-u.ac.jp

*1 http://heartbleed.com/

from its respective software ecosystem - Node Package Manager (npm)*2. We will be presenting two of these projects to showcase our results*3. Finally we evaluate the tools general comprehension and understanding through an on-line survey.

2. Research Background

In this section we introduce the previous research conducted regarding library updates and dependency updating.

2.1 Research Body

Software visualization has grown in the past years and branched out steadily into its own separate research field and has been recognized by international conferences. This is due to the ever-growing research body directed at creating various tools to assist software developers. There isn't yet a visualization that is able to incorporate all the needs of the modern day software development process. Because of the gravity of the data required to be visualized each visualization can only address a certain amount of points. The majority of tools created are targeted at helping developers to maintain their projects, easily understanding large scale projects or even assisting newly recruited developers to adapt with a project, learn and understand the logic behind the code.

Code Bubbles [13] is a tool that organizes the source code into interrelated edible fragments (bubbles) integrated onto the user interface. Another tool visualizes the source code as a Code City [14]. Using a 3D city-like structure it shows the relative complexity of modules within the code. Finally Telea *et al.* presented a tool called Code Flows [15] which highlights software structural evolution.

Naturally, there is research conducted with the focus on libraries, not at the code as whole. Library migrations and updated have been addressed in [16] and [17]. Their results showcase the challenges present in updating libraries. Furthermore, Mileva *et al.* [18] visualized popularity trends of a single open source library.

This paper follows-up the visualization work conducted at our laboratory. Yano *et al.* created VerxCombo [19], which is a tool that assists developers in making library maintenance decisions. The tool was able to show the usage between different versions of libraries amongst developers in the ecosystem, but was limited to only showing maximum 3 at a time.

A tool from Kula *et ak.* [20] uses a circular visualization method and includes systems and their dependent libraries while incorporating historical data. The flaw of that tool is that it wanted to show too much and was formidable for inexperienced users. Based on these previous works we created our visualization tool, which uses an intuitive visualization and strives for balanced between relevant information and visibility, while adopting a new popularity measurement unit.

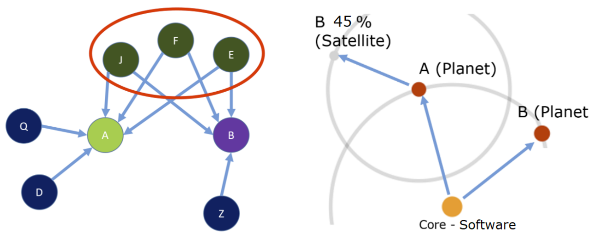
2.2 Definitions

For the purpose of this paper we define the following terminologies as follows:

(1) **Software** - program, whose dependencies to other libraries are our concern for visualization.
(2) **Software Library** - the programs used by the software or other libraries, forming library dependency directed graph.
(3) **Visualization** - the representation of an object and set of information on a chart, plot or other image

3. Coexistence Coefficient

Coexistence coefficient is defined by Kula *et al.* [12] as - *"Coexistence pairing examine and explores occurrences of specific combinations between software components"*. In this paper it is depicted as a binary relation between a pair of libraries (2) within a specific software and its ecosystem, where those two libraries are used by at least one common library or common software.



(a) Coexistence Logic (b) Mapping the Coexistence

Fig. 1: Library Coexistence Mapping

3.1 Coexistence Logic

To further elaborate the logic behind the coexistence coefficient we present an example of coexistence between libraries. Based on the previous sections definition we know that coexistence is between a pair of libraries, therefore, we direct the attention to Fig. 1a where the pair is represented by *Library A* and *Library B*. Similarly, *Q, D, J, F, E, Z* are also libraries within the same language ecosystem. These five libraries collectively use the pair in focus, depicted by the directed edges. In this example, *Library A* is used by *Q, D, J, F, E* and *Library B* is used by *J, F, E, Z.* We denote this usage as: $UsersA = \{Q, D, J, F, E\}$ and $UsersB = \{J, F, E, Z\}$.

Fig. 1a depicts the second part of the coexistence coefficient definition, which is the common combination occurrences. In detail, both *Library A* and *Library B* have common users - *Library J, F and E*. These three users reference the library pair in their software code. We denote these common users with the following notation: $UsersA \cap UsersB = \{J, F, E\}$. We also define coexistence coefficient (cc) of A for B as follows:

$$CC_A(B) = \frac{|UsersA \cap UsersB|}{|UsersA|} \tag{1}$$

meaning, Library A's cc for Library B's is the ratio of A's users which are simultaneously B's users.

Next, we demonstrate the equation with a sample data set. We select two libraries from the npm ecosystem - *babel-core* and *mocha*. The respective total users are - *babel-core* - 6,884 and *mocha* - 3,165*4. The two libraries share a coexistence relation,

*2 https://www.npmjs.com/
*3 full result page at: https://goo.gl/AxkUsR
*4 as of July 2017, dataset collection period

because they share 3,158 common users. Applying these sample numbers to equation 1 we have the following model:

$$\begin{aligned} CC_A(B) &= \frac{6884 \cap 3165}{6884} \\ &= \frac{3158}{6884} = 0.4587 \end{aligned} \tag{2}$$

The result from equation 2 shows that the cc between babel-core and mocha is 0.4587 (45.87%), meaning that almost half of *babel-core's* users are also *mocha's* users. On the other hand, *mocha's* cc for *babel-core* is 0.9978 (99.78%), concluding almost all of *mocha's* users are also *babel-core's* users.

3.2 Library Coexistence Mapping

In order to keep things consistent and to avoid redundancy, we will explain briefly how the coexistence is represented onto an orbital setup. The detailed explanation and logic will be explained thoroughly in the following section.

Fig. 1b depicts a small example of an orbital layout, including a core element (software), planet element (library) and a moon (coexisting library). In this example the software name is not of relevance, but it is a software that uses two libraries. We take the two libraries from the coexistence logic section explanation. In this case, Library A becomes *babel-core* and Library B - *mocha*. Both are used by the software and are plotted as a planet element, directly connected to their parent - the Core (Software). In order to illustrate the binary relation between both of them we make use of the moon element. The moon is a direct representation of the result from equation 2. The moon and planet hold essentially the same name - *mocha* but fundamentally hold a different meaning. As a planet, *mocha (B)* is a library used by the software, while as a moon of *babel-core (A)*, it is depicted as a coexisting library with its parent planet.

4. Tool Concepts

In this section we show and explain how both the orbital layout and the coexistence coefficient fit together.

4.1 Solar System Metaphor

To achieve our goals for the tool, we had to select the correct visualization. The solar system visualization metaphor, served as the correctly corresponding analogy, due to it being similar to a software system. A software system, contains a core that binds all the elements together, i.e. the software itself. The elements it uses in terms of foreign objects (libraries) are "attracted" by it and are in its orbits. Accordingly, the similar analogy applies for the coexistence between the libraries within the software, but their duality severs to represent different information.

4.2 Visualization Design and Representation

In order to read, comprehend and evaluate the results of our visualization, we firstly defined the data element in previous sections. Here we define individually the visual elements followed by an example.

- **Core (Sun)** - being a center of a solar system, holds similar meaning within our visualization. It denotes the software, which uses libraries within its structure. The color of the sun element is orange and is unique within the visualization. For each separate software, which has a SoL Mantra created for it, the name of the software will be written in the center and denoted by a sun element.
- **Planets** - are the direct representation of the libraries used by the software. Each planed that orbits the sun, denotes a distinct object and are not repeated as such. In detail, if a library uses 6 libraries, there will be a total of 6 planet visualization objects plotted in orbits around the sun.
- **Color** - serves to denote if a library used by the software is up-to-date or outdated. Each planet element will be highlighted with either a red color, i.e. the library is outdated, or a green color - up-to-date. Planets can not be both or have other colors filling their element.
- **Rotation** - assists the color element. There are two states: rotating and static.The static planets, also highlighted in green color, are libraries that are up-to-date, while the rotating, filled with red, planets are outdated. The goal is to make the outdated count of libraries easily visible, while making the up-to-date libraries more passive and "safe".
- **Moons** - represent the coexisting libraries. Moons are created after the coexistence coefficient is calculated for the library pairs in the software. Only planets are able to have moons. If a planet has no moons (0) it means it has no coexisting library with it within the software, although it can be a moon of another library (planet). Moons are duplicates to the libraries that the software uses with the purpose of showing which libraries should be considered to be updated together. In detail, if a planet that has a moon or moons is denoted with a red color, requires examination of its coexisting elements to evaluate the update complexity and potential candidates to be updated together.

Table 1: *Ranza* Dependencies

Library	Coexistent Library (*cc*)
mocha	None
supports-color	mocha(99.09%)
glob	mocha(23.05%), supports-color(1.59%)
char-spinner	glob(100%), mocha(100%), supports-color(100%)
bluebird	char-spinner(0.2%), glob(49.49%), mocha(12.32%), supports-color(0.85%)
babel-core	bluebird(95.45%), char-spinner(0.76%), glob(95.45%), mocha(45.87%), supports-color(3.17%),

4.3 Illustrative Example

Through an example visualization, we will demonstrate each representation. To this end, we will use a sample package from the npm repository - *ranza*[*5], a dependency checker which depends on 6 libraries as shown in Table 1 - *mocha, supports-color, glob, char-spinner, bluebird, babel-core*. The table also shows each libraries coexisting library within the software and the corresponding calculated cc percentage.

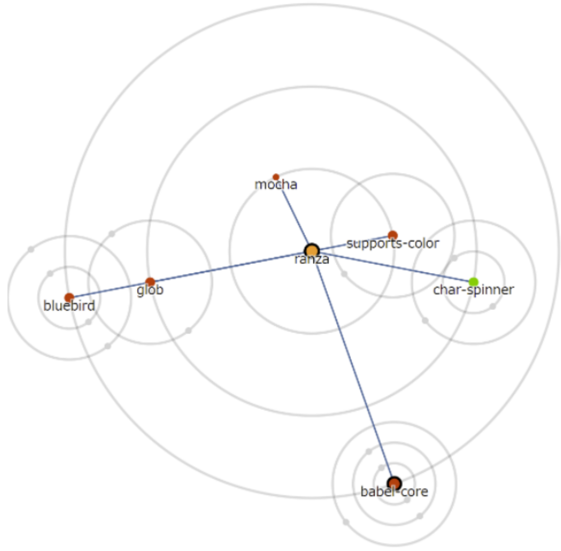
Fig. 2a illustrates the SoL Mantra of the *ranza*[*6] package with

*5 https://github.com/raphamorim/ranza
*6 data from July 2017

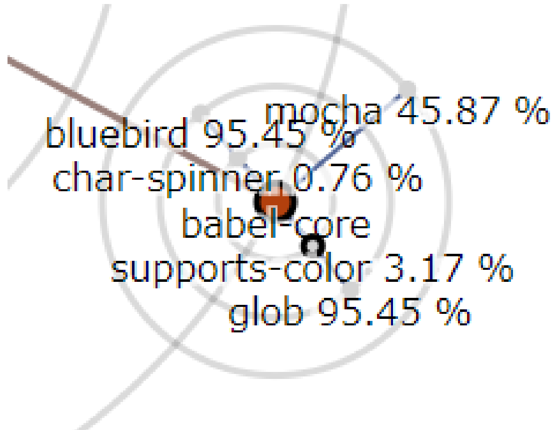
all the before mentioned visual representations present. In the center stands the software as the sun element. All 6 library dependencies are plotted as individual planets orbiting the core with their respective binary coloring schema. Only one library is in green, which means that only *char-spinner* is the latest version available. Accordingly, the remaining ones are outdated, and the planet elements are rotating around their orbits*7.

Next up, we navigate with the cursor to observe the coexistence between the libraries. Fig. 2b depicts a highlighting event when the mouse cursor is over a specific planet. In the example shown, the library in question is *babel-core* and it coexists with all the remaining libraries in the *ranza* package. Individual coexistence coefficients are located besides the coexisting library and rounded up to two decimals.

The coexistence coefficient percentage, holds two meanings. Firstly, it denotes that there is a binary relationship between the given pair as explained. Secondly, it shows the magnitude of the coexistence, meaning that how often the pair are used together. This provided information serves to evaluate the potential risks that are present while considering *babel-core* for an update within the *ranza* software. For example, *babel-core* and *bluebird* have a cc of 95.45%, meaning that almost all of *babel-core's* users are also *bluebird's* users. This usage suggest that these libraries work well together and should be considered for a simultaneous update. Similarly, *mocha* has a cc of 45.87%, therefor the majority of users, have the library pair in their source code. Although not as strong cc as the previous example, it still should influence a careful consideration while updating.



(a) Ranza



(b) babel-core

Fig. 2: Ranza SoL Mantra

5. Evaluation

In order to evaluate the tool, firstly we gathered a sample data set and created a total of 23 distinct visualizations of the corresponding software packages. Here we also show how our tool works on real world projects. Based on the collected data we perform an empirical study of the 23 packages and show case 2 of them as an example and proof of concept. Continuing, after successfully conducting the real world project experiments, we generated an on-line survey with the goal of evaluating our tools comprehension.

5.1 Experiments

First part of our tool evaluation consists of a creating visualizations and analyzing the collected software packages from the npm repository.

5.1.1 Dataset

Since the tool mostly consists of JavaScript, we decided to apply it to other projects within the ecosystem. The JavaScript repository hosts over 230,000 packages with new ones being constantly added[21]. To this end we selected one of the most popular projects for 2016 amongst developers*8. Initially there were 30 projects but because some of them had no library dependencies or less than 2, we had to discard them. This resulted in having a dataset of 23 packages from the npm ecosystem. Fig. 2 shows the general data and the names of the projects tested. We show that through the mean and median of the stars*9 that the projects are indeed popular. Furthermore, with the average commits, issues, contributors and releases we can say with certainty that the projects are still active and being worked on. Finally, we can see that most of them have a relatively high number of other depending packages, with the exception of *pm2* which has only 330.

We implemented a script based tool to gather and collect the data needed for the visualization. In order to generate our data and calculate cc, we used packages from the npm repository. We relied upon packages such as: *is-outdated* package*10 to detect if the libraries are outdated or not. We used the output of it to compare with the package.json file available in the local folder. If the version matched, we considered them up-to-date and alternatively, if they do not, we apply the outdated tag. For the coexistence coefficient, we used several packages in unison. We began with generating the users for every package using *get-dependencies* package*11. After that we created the pairing sets of libraries using *js-combinatorics* package*12. Lastly, we extracted

*7 not visible on a static image
*8 experiments conducted July 2017
*9 github ranking metric
*10 https://github.com/rogeriopvl/is-outdated
*11 https://github.com/SharonGrossman/get-dependencies
*12 https://github.com/dankogai/js-combinatorics

Table 2: Collected Data Summary

Data	Stars	Pull Requests	Issues	Commits	Contributors	Releases	Branches	Dependencies	Dependents
Minimum	917	0	0	186	15	11	1	0	330
Maximum	67,706	180	690	8640	1595	396	103	47	24,432
Mean	9914	30	149	1492	158	77	17	9	4272
Median	11,863	21	168	2162	177	89	14	7	2279

Tested Packages: express, request, browserify, grunt, pm2, socket.io, mocha, gulp-uglify, cheerio, passport, hapi, react, karma, pug, mysql, less, mongodb node.js driver, jshint, morgan, webpack, restify, magick, jsdom

the common users between each set using *comparray* package[*13] and calculated the final cc score.

In this paper we will demonstrate two distinct examples from our dataset in order to illustrate the use of our visualization and how a developer would use our tool and the information provided. The first example is *react*[*14], which is the highest starrted package amongst the 23 packages in the dataset. For the second example, we will use *cheerio*[*15] to showcase how our tool performs on larger systems.

5.1.2 Empirical Study on Real World Projects

Here we demonstrate our tool applied to two real projects.

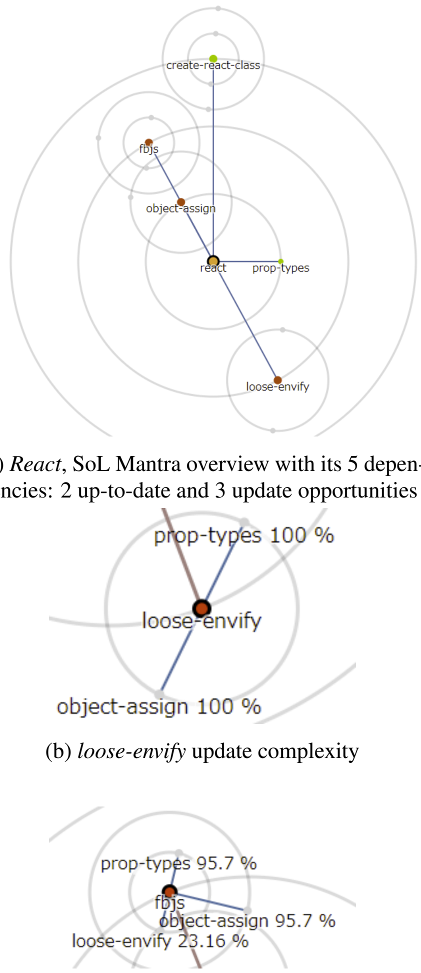
1. Example - *react*

React is a popular library used for interface creation developed and maintained by Facebook.

From Fig. 3a we can see the generated SoL Mantra overview for the *react* package. As explained in Section 4 the software is represented by the core element in orange. The 5 dependencies the software uses are represented through the 5 planets orbiting the core - *fbjs, object-asign, create-react-class, prop-types, loose-envify*. Continuing, 2 of those are colored green, indicating that they are up to date[*16] with the latest version available on the npm repository[*17]. Alternately, the remaining three, colored in red, libraries are outdated and should be considered for an update.

We focus our attention towards the outdated libraries - *fbjs, object-assign, loose-envify* which present an update opportunity and proceed to evaluate the related update complexity through the visualization elements. Using the cursor we are able to navigate through the tool and focus each planet element individually. Hovering over it, highlights its moons and shows the coexistence coefficient that they have with their parent.

Firstly we check *loose-envify*. The library has 2 coexisting libraries within the *react* package (Fig. 3b) and both have 100% cc score. This shows us that these libraries are used all of the time together within the npm ecosystem and therefor have strong coexistence. With such high cc score we can safely say that when deciding whether or not to update *loose-envify*, we must consider its coexisting libraries for an update, otherwise risking potential failures, incompatibilities. This case presents a simple decision because of the high score present, although our visualization doesn't tell the developers that they have to update. The decision ultimately depends on the developer and other factors.



(a) *React*, SoL Mantra overview with its 5 dependencies: 2 up-to-date and 3 update opportunities

(b) *loose-envify* update complexity

(c) *fbjs* update complexity

Fig. 3: *React* SoL Mantra with 3 complex update opportunities

Next we have *fbjs* (Fig. 3c which has 3 coexisting libraries - *loose-envify, object-assign, prop-types*. We see that *fbjs* has high cc score with two of them - 95.7%. Although not 100% like the previous case, the score remains on the high end. Based on that we apply the same reasoning as *loose-envify* and conclude that if an *fbjs* update is considered, so must be *object-assign, prop-types*. In this example *prop-types* is already up-to-date and could be ignored while evaluating the complexity. Lastly, we have a 23.16% cc with *loose-envify*. This cc score is particularly low but still means that 1 in 5 *fbjs* users use both libraries together. Additionally, with *loose-envify* being an update candidate, and considering that *fbjs* coexists with it, shows the entangled complexity that exists between libraries. Following that, we should re-evaluate the update complexity of *loose-envify*.

[*13] https://github.com/JonathanPrince/comparray
[*14] https://github.com/facebook/react
[*15] https://github.com/cheeriojs/cheerio
[*16] as of July 2017
[*17] latest main version, not including beta versions

The last outdated library is *object-assign*[*18]. It has only one coexisting library - *prop-types* with 40.11% cc. Because *prop-types* status we can safely update this package's version, as long as, it is the only one considered for an update. Arguably, if we have to consider the entire system for an update, through the cc scores we can concretely deduce that all of the outdated libraries should be updated together.

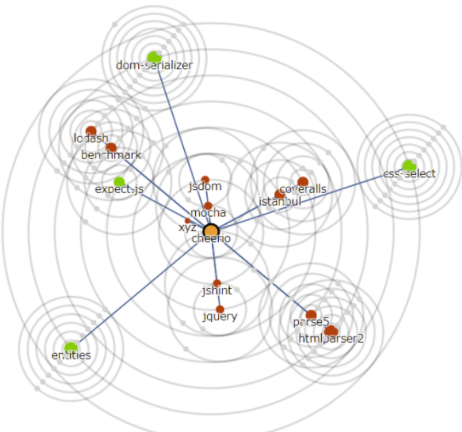


Fig. 4: *Cheerio* SoL Mantra - 11 outdated and 4 up-to-date libraries.

2. Example - *cheerio*

For the second example we will demonstrate how a software with a lot of libraries is represented. To this end we selected the *cheerio* package which provides implementation of core jQuery[*19] for servers.

Cheerio compared to the previous example, references more library packages in its code. Fig. 4 shows the generated SoL Mantra overview of the package. *Cheerio* uses 15 libraries in total, 11 outdated and only 4 up-to-date[*20]. The up-to-date libraries in the package are *expect.js, entities, dom-serializer and css-select*. All of them have high moon count, but because the package is already using the latest version of them, we can safely continue to evaluate the complexities of the remaining 11. In cases like *cheerio*, where the number of used packages and cc between them is abundant, even the up-to-date libraries and their cc's could be evaluated while making update decisions.

Continuing with the outdated ones, we are unable to show each and everyones cc and their scores but we will summarize our findings. The full example could be seen on the tool's page. Out of the 11 there are 6 libraries with comparatively high coexistence count than the others - *istanbul, coveralls, parse5, htmlparser2, benchmark and loadash*. In detail *istanbul* has 5 libraries with 3 of them having a cc score of 92.87%, one with 69.53% and the least having 1.16%. *Coveralls* with 7 coexisting libraries is a similar case as *istanbul*. Out of the 7, there are 5 that have a cc score of 96.18% and similarly 69.92% followed by 3.66%. The remaining ones also follow the same pattern, therefore we will not be listing them as detailed.

5.2 Survey Evaluation

With the examples we successfully demonstrated how through cc we were able to show how each libraries are used together. To further our evaluation, we tested through an on-line survey the usage comprehension and readability of our tool, while in the hands of people that have never used it before. The survey was divided in 4 sections - *pre-task questions, general comprehension, coexistence coefficient comprehension, post task questions*.

1. Pre-task questions

Here we requested general information of the survey users, such as age, affiliation and prefer or used language in their environment. More than half of the users are aged 25-30 and around 30% are 18-24 years old. Most of them are still in the university, although we managed to receive one sibmission from a professional working in the field of computer science. Lastly the most used coding language amongst the participants is split between Java and Python followed by C and C++.

2. General Comprehension

In this part of our survey we wanted to evaluate how easy it is for a new users to handle our tool while providing them with only limited information and general legend on how to read the tool. Firstly they had to answer which is the software represented, to which all of them answerd with 100% accuracy. The last two questions were focused on library usage information. Almost all gave correct answers regarding number of used libraries and outdated ones.

Based on the results from this section we conclude that the basics visualization concepts of our tool are easy to understand for people without much software development experience, hence it should be also easy for experienced developers. In detail, the high answer accuracy suggests that the general information is easy to understand and see.

3. Coexistence Coefficient Comprehension

This section is divided into two parts with the first part aiming to see if people understood the coexistence coefficient and how to generally read it. The second part's goal is to assess the update complexity comprehension with our visualization. We gave our definition for coexistence and asked the participants if they understood it and most asnwered yes. Following that, we asked them with multiple choices, which libraires didnt have any coexisting ones within the system. The majority (63.2%) answered this question correctly, with 21.1% giving a partially correct answer. Lastly, we asked them to tell us what is the highest cc count in this software. Overwheming part of the participants also gave a correct answer to this question - 78.9%. Here we can conclude that the general idea for coexistence is easy to understand in a short time, while exploring the tool, provided the definition is present.

To understand if the update complexity is easily comprehendable we presented the users with 4 cases and asked them on a scale from *Further investigate (1)* to *Safe to update(5)* seperately.

Table 3 shows the distinct cases with the list of coexisting libraries. Case 1 and 2 as shown have no coexisting libraries. We asked the users to asses the update complexity of *mocha and*

*18 not shown with a figure
*19 https://jquery.com
*20 as of July 2017

Table 3: Update Complexity Comprehension Cases Information

Case	Library	Coexisting Libraries
1	mocha	None
2	eslint-plugin-chai-friendly	None
3	bluebird	eslint-plugin-chai-friendly(0.08%), safe-buffer(1.56%), chai-as-promised(4.37%), lodash.isplainobject(5.75%), qs(13.63%), chai(30.28%), eslint(41.33%), mocha(48.9%)
4	eslint	eslint-plugin-chai-friendly(0.24%)

eslint-plugin-chai-friendly to observe how the participants would evaluate the risk based on the coexistence coefficient definition provided.

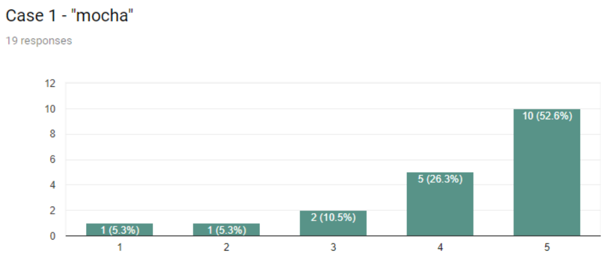


Fig. 5: Case 1 result summary

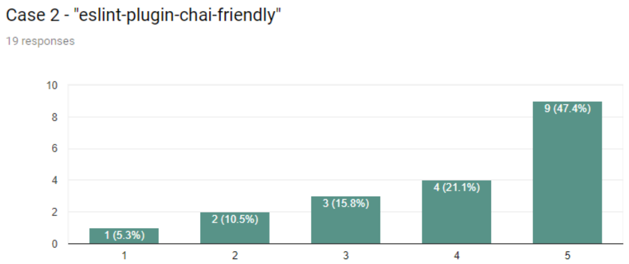


Fig. 6: Case 2 result summary

Mocha has no coexisting library, therefore on the mentioned scale the absolute correct answer is "Safe to update(5)". As seen on Fig. 5, 10 of our participants answered correctly, followed by 5 giving a slightly lower answer on the scale, which is still considered accurate. The remaining 4 answers are considered wrong in this case.

Similarly, the second case also included a library with zero dependencies. We can see from Fig. 6 that the answers are similar with the first case. Based on the results we can conclude that when there are no coexisting libraries it is easy for new users to assess the risks when using the SoL Mantra tool.

For the third case we presented them with the library with the highest cc count in the example visualization - *bluebird*. The coexisting libraries and their respective cc score can be seen in Table. 3. For this case, the expected answer is "Further investigate" in terms of making an update decision. The participants properly understood the update complexity and 15 of them gave the correct answer, as seen on Fig. 7. The result could be interpreted in two ways: either the participants really understood the complexity or because it had a lot of moons they decided it must be the correct answer. Both assessments are generally correct, but the important thing is that our users aren't specialists.

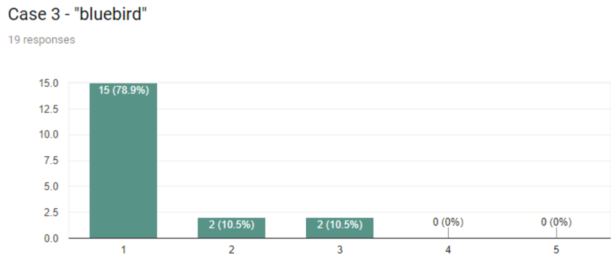


Fig. 7: Case 3 result summary

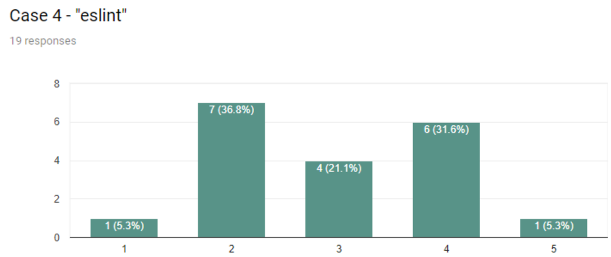


Fig. 8: Case 4 result summary

For the last case, we provided a library with only 1 dependency. We wanted to see how will the participants evaluate such a case. Fig. 8 highlights the results from the survey. The *eslint* has a cc score of 0.24% with its coexisting library. Because of that it isn't considered a high complexity update opportunity, if evaluated on its own. Due to the nature of this case, the expected answer is not a strong 1 or 5 from our scale. Accordingly, the majority of the participants also evaluated the case the same way as us. Although the absolute accurate answer is a 3, the adjacent ones are also acceptable.

To conclude, amongst our survey users, the majority managed to accurately grasp the concept of coexistence coefficient applied to our visualization and made accurate update complexity conclusions.

4. Post Task Questions

Here we asked the survey users to give us their feedback after experiencing the SoL Mantra tool and answering the comprehension questions. When asked if they found the tool useful or potentially useful, all but one answered with "yes".

Next we wanted to see what the participants thought how the tool feels when using it. Fig. 9 visualizes the results provided with a scale from *Natural(1) to Counter-intuitive(5)*. Most of our participants thought the tool feels adequate when using it, while some thought its difficult. This shows us that there is still room

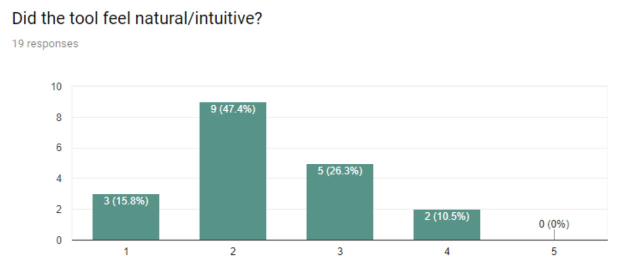


Fig. 9: Post Task Question Results

for improvement in order to be easily accessible for everyone.

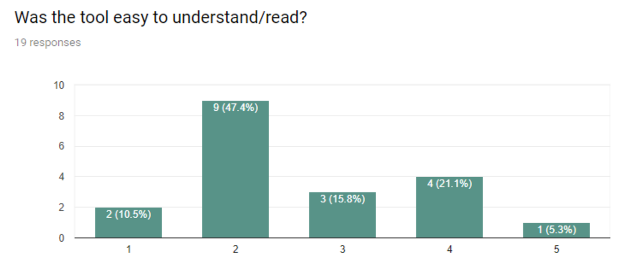


Fig. 10: Post Task Question Results

The last question we asked was if the participants found the tool easy to comprehend. From Fig. 10 we can see that although more than half stated on a scale from *Very easy (1) to Extremely difficult/confusing(5)*b that it is easy, the remaining did indeed find some difficulties with it. As one of the goals for the paper is to created a tool that is easy to use while providing sufficient information for library update complexities, we can derive from these results that there is room for improvement.

Overall, we consider the survey to be successful and achieved its goals. All of the participants agreed to our electronic consent before doing the survey, and provided us with a much needed feedback.This survey ran from the start of January 2018 until the first week of February 2018 and only produced 19 results.

6. Conclusion and Threats to Validity

In this paper we present the SoL Mantra tool that we developed using the orbital layout visualization concept together with the coexistence coefficient metric. We stated our decision process while selecting the visualization concept and added the benefits it provides, while also weighting in the drawbacks. Although a new metric, the coexistence coefficient provided a great solution to address a greater scope of library interdependency. Combining both together felt natural and we manage to provide relevant information for library update opportunities and their respective complexities to software developers.

First and foremost a potential risk to our results is the coexistence coefficient metric. Furthering, the results produced by the coefficient may not be generally applicable to all softwares and all ecosystems. Our collected data is based on established projects that have been developed by dedicated teams, hence, we have not tested the tool's performance with smaller or personal projects. Furthermore, the projects we tested, could have specific library version for internal or other reasons. Another threat is that we only evaluate the data provided by the npm repository and we don't consider if the library reference within the code has been used or just referenced.

Acknowledgments This work is supported by JSPS KAKENHI (Grant Numbers JP25220003). Special note to Kaoru Ito, who assisted and giving the tool's name.

References

[1] Ebert, C.: *Open Source Software in Industry*, in IEEE Software, Vol. 25, No. 3, pp. 52–53, 2008.
[2] Cadariu, M. and Bouwers, E. and Visser, J. and van Deursen, A.: *Tracking known security vulnerabilities in proprietary software systems*, 22nd IEEE SANER, pp. 516–519, March 2015.
[3] Xia, P. and Matsushita, M. and Yoshida, N. and Inoue, K.: *Studying reuse of out-dated third-party code in open source projects*, Information and Media Technologies, Vol. 9, No. 2, pp. 155–161, 2014.
[4] AGerman, D. M. and Gonzalez-Barahona, J. M. and Robles, G.: *A model to understand the building and running inter-dependencies of software*, 14th WCRE, pp. 140–149, Oct. 2007.
[5] YKula, R. G. and German, D. M. and Ouni, A and Ishio, T. and Inoue, K.: *Do Developers Update Their Library Dependencies?*, Empirical Software Engineering, 2017.
[6] Bavota, G. and Canfora, G. and Di Penta, M. and Oliveto, R. and Panichella, S.: *How the Apache Community Upgrades Dependencies: an Evolutionary Study*, Empirical Software Engineering, Vol. 20, No. 5, pp. 1274–1317, 2015.
[7] Bogart, C. and Kästner, C. and Herbsleb, J.: *When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies*, ASE (Workshop SCGSE), pp. 86–89, Lincoln NE, 2015
[8] Raemaekers, S. and Deursen, A. van and Visser, J.: *Semantic versioning versus Breaking Changes: A Study of the Maven Repository*, 14th IEEE SCAM, pp. 215–224, 2014.
[9] Kula, R. G. and German, D. M. and Ishio, T. and Inoue, K.: *Trusting a Library: A Study of the Latency to Adopt the Latest maven Release*, 22nd IEEE SANER, Vol. 22, pp. 520–524, Montreal, Canada, 2015.
[10] McIntosh, S. and Adams, B. and Nguyen, T. H. and Kamei, Y. and hassan, A. E.: *An empirical study of build maintenance effort* 33rd ICSE, pp. 141–150, DOI: http://doi.acm.org/10.1145/1985793.1985813, New York, NY, USA, 2011
[11] Foundation, O.: *Top 10-2017 Security Risks*, OWASP, available: https://goo.gl/fCs5ob, 2017.
[12] Kula, R. G. and Roover, C. De and German, D. M. and Ishio, T. and Inoue, K.: *Generalized Model for Visualizing Library Popularity, Adoption and Diffusion within a Software Ecosystem*, 25th IEEE SANER, Campobasso, Italy, March 2018 (to Appear).
[13] Bragdon, A. and Reis, S. P. and Zelezni, R. and Karumuri, S. and Cheung, W. and Kaplan, J. and Coleman, C. and Adeputra, F. and LaViola Jr, J. J.: *Code bubbles: Rethinking the user interface paradigm of integrated development environments*, 32nd ACM/IEEE ICSE, Vol. 1, pp. 455–464, New York, NY, USA, 2010.
[14] Wettel, R. and Lanza, M.: *Visually localizing design problems with disharmony maps*, in Proceedings of the 4th ACM Symposium on Software Visualization, pp. 155–164, New York, NY, USA, 2008.
[15] Telea, A. and Auber, D.: *Code flows: Visualizing structural evolution of source code*, Computer Graphics Forum, Vol. 27, No. 3, pp. 831–838, 2008.
[16] Teyton, C. and Falleri, J. R. and Palyart, M. and Blanc, X.: *A study of library migrations in Java*, Journal of Software: evolution and Process, Vol. 26, No. 11, 2014.
[17] Bauer, V. and Heinemann, L. and Deissenboeck, F.: *A structured approach to assess third-party library usage*, 28th IEEE ICSM, pp. 483–492, Sept. 2012.
[18] Mileva, Y. M. and Dallmeier, V. and Zeller, A.: *Mining API popularity*, TAIC PART, pp. 173–180, 2010.
[19] Yano, Y. and Kula, R. G. and Ishio, T. and Inoue, K.: *VerxCombo: An interactive data visualization of popular library version combinations*, in 23rd IEEE ICPC, pp. 291–294, Florence, Italy, 2015.
[20] Kula, R. G. and Roover, C. De and German, D. M. and Ishio, T. and Inoue, K.: *Visualizing the Evolution of Systems and Their Library Dependencies*, in 2nd IEE VISSOFT, pp. 127–136, Victoria, BC, Canada, 2014.
[21] Wittern, E. and Suter, P. and Rajagopalan, S.: *A Look at the Dynamics of the JavaScript Package Ecosystem*, 13th MSR, pp. 351–361, Austin, TX, USA, 2016.

