SoL Mantra: Visualizing Update Opportunities Based on Library Coexistence

Boris Todorov*, Raula Gaikovina Kula†, Takashi Ishio†, Katsuro Inoue*
*Osaka University, Osaka, Japan
†Nara Institute of Science and Technology, Nara, Japan
Email: {boris-t, inoue}@ist.osaka-u.ac.jp, {raula-k, ishio}@is.naist.jp

***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 become difficult to manage and avoid compatibility issues or bugs. 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 large software with a lot of dependencies. To address those difficulties, we propose SoL Mantra which is a tool that shows update opportunities by leveraging the Wisdom of the Crowd in a software ecosystem. Using this combined knowledge, our tool displays 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 two examples from the JavaScript ecosystem. Through these examples, we demonstrate how maintainers can benefit from SoL Mantra's visual cues.**

I. Introduction

For the past decade, predominant practice within software engineering has been the usage of third-party software, also known as software library [1].

The merit of using software libraries lies in the reusing of code. The inherited benefits include reducing the man-hours cost when developing new software, safety and stability from codes approved by various developers. However, as time passes, the libraries grow older and newer versions are released, providing security improvements, new features and bug fixes. That also brings risks for software maintainers, whose job is to keep a software running and operational throughout its life cycle. Updating a library dependency could bring all the beforehand mentioned improvements, but can also cause devastating problems to the software.

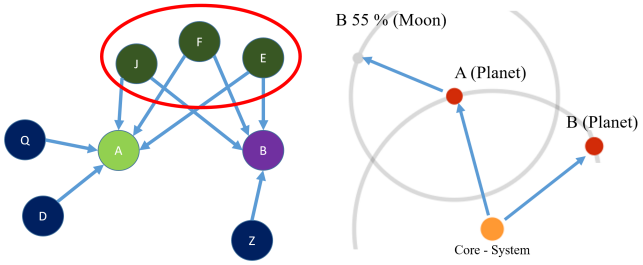
Researchers have observed and document empirically how software developers interact and deal with library updating. Some even conclude based on a large sample of Java clients that use Maven libraries, that high number of systems keep their dependencies outdated [2]. Others provide in-depth analysis on the impact of newer version releases and what changes within them drive developers to adopt newer versions [3], [4].

Software maintainers, have to carefully evaluate various risks while making decisions "if" and "when" to update their software's library dependencies. Studies have been conducted expressing concerns in incompatibility when updating [5], [6]. Oppositely, not updating a library could also lead to devastating problems, such as the heartbleed bug[1]. Libraries 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, their corresponding versions and supported documentation.

Previous works have tackled the problem of providing a visual aid for update opportunities. Kula et al. [7] visualized the evolution of systems and their dependencies. While this approach uses historical data to determine library update candidates, we apply the combined developer knowledge in the entire ecosystem. Yano et al. [8] proposes VerXCombo, a prototype tool, that presents various library version combinations. Similarly to our work, VerXCombo uses Wisdom of the Crowd to provide library combinations and provides more detailed information.

In this work, we address the problem with updating libraries and propose the peculiar Software Library Mantra tool (SoL Mantra), applying the combined knowledge of all software developers of a language ecosystem (Wisdom of the Crowd) with coexistence coefficient (*cc*), defined by Kula et al. [9] and described in detail in Section II. We provide 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. We demonstrate the usefulness of our tool, by using two visualization examples based on a data sample of popular libraries within the JavaScript ecosystem. The first one consists of a smaller system, while the second one depicts a larger one. In both cases, our tool contributes and provides update opportunity information. With them, we will also display how our tool helps to identify update opportunities and using *cc* understand the complexities for each candidate. A demonstration with 23 library examples can be observed at https://goo.gl/2Rewn4

[1] http://heartbleed.com/



(a) Coexistence Logic (b) System that uses A and B

Fig. 1: Library Coexistence Mapping

II. Tool Basic Concepts

Our main objective is to provide an intuitive visual tool to assist maintainers by providing update opportunities information. The SoL Mantra, checks if a system's dependencies are up-to-date and by using a coexistence notation to determine the complexity of each potential update.

A. Solar System Metaphor

Our selected layout for the visualization is the Orbital Layout[2]. This visualization was considered after investigating the *Hexagonal Binning*[3] and Dependency Wheel visualization technique[4]. This visualization uses the D3.js API[5].

B. Basic Data Concepts

In order to understand the visualization, we define the following data elements:

- ***System*** - the program, whose dependencies to other libraries are our concern for visualization.
- ***Library*** - the programs used by the system or other libraries, forming library dependency directed graph.
- ***Coexistence(cc)*** - as defined by Kula et al [9], coexistence is a binary relation between two libraries (nodes) in an ecosystem, where those two libraries are used by at least one common library or common system.

Fig. 1a is an example of coexistent of libraries. For instance, library A is used by Q, D, J, F and E, respectively, library B by J, F, E and Z. We see that there are three users that use both A and B - J, F and E or we denote: $UsersA \cap UsersB = \{J, F, E\}$, where $UsersX$ denotes the set of users of X. We also define the coexistence coefficient (cc) of A for B as follows:

$$\frac{|UsersA \cap UsersB|}{|UsersA|} \quad (1)$$

meaning, A's cc for B is the ratio of A's users which are simultaneously B's users. For example, libraries *babel-core* with 6,884 users, and *mocha* with 3,165 users in npm ecosystem are in coexistence relation, because they share 3,158 common users, thus the cc of *babel-core's* for *mocha* is 0.4587 (45.87%), meaning about half of *babel-core's* users are also *mocha's* users. On the other hand, *mocha's* cc for *babel-core* is 0.9978, meaning almost all of *mocha's* users are also *babel-core's* users.

[2]https://github.com/emeeks/d3.layout.orbit
[3]https://github.com/d3/d3-hexbin
[4]http://www.redotheweb.com/DependencyWheel/
[5]https://d3js.org/

TABLE I: *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%),

C. Visualization Design and Representation

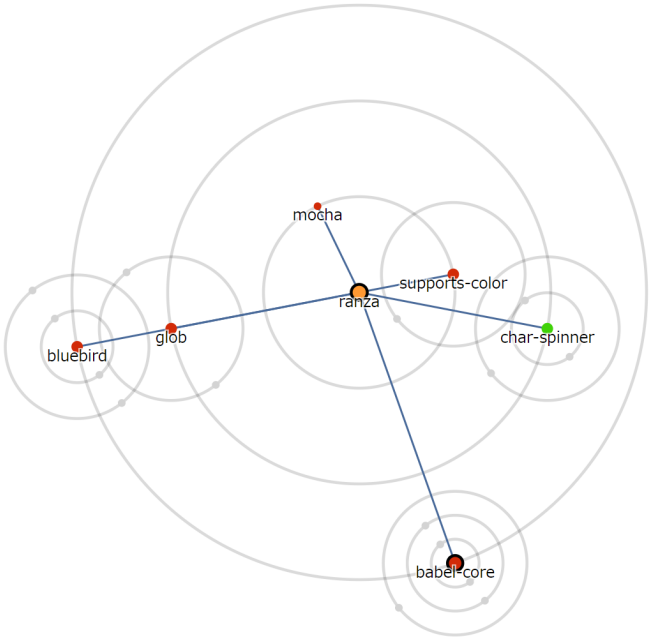
Fig. 1b shows our visual design with the system always represents the core of our orbital layout and is denoted by the orange color (sun). All the libraries used by the system are denoted as planets orbiting the system, hence every library will be a separate planet regardless of its coexistence with other libraries. Coexisting libraries will always be orbiting around the library (planet) they coexist with. To further elaborate on the matter, library B is both a planet referenced in the system and a coexistent (moon) with library A.

To vividly explain our visualization and its functionalities we will use a real example with a sample package and thoroughly explaining each of the visual elements and how they work. The package in question is *ranza*[6], a dependency checker package which contains only 6 dependencies, as shown in Table I.

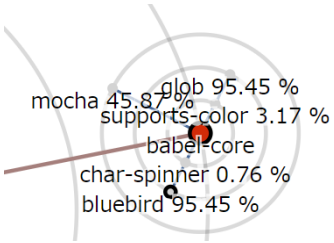
Fig. 2 illustrates the SoL Mantra for the *ranza* system example and the various elements we created:

- ***Core (Sun)*** - in the center of the visualization, lies the software system that uses libraries. Fig. 2a shows the *ranza's* system Sol Mantra, therefore *ranza* is placed as the core (yellow node).
- ***Planets*** - the planets are direct representations of the libraries used by the software system. For example on Fig. 2a, there are 6 planets showing the 6 libraries that *ranza* uses - *char-spinner*, *supports-color*, *glob*, *mocha*, *bluebird* and *babel-core*. The distance from the Sun currently has no meaning, apart from increasing the readability.
- ***Color*** - every planet has an outdated flag, based on a boolean value. The color filling the planet changes to represent if the system uses the latest version or not, i.e. green - up-to-date and red - outdated.
  As seen on Fig. 2a *char-spinner* is green, which denotes it is up-to-date, while *supports-color*, *babel-core*, *mocha*, *glob* and *bluebird* are red, therefore update candidates.
- ***Rotation*** - to make sure outdated packages are easy to spot, we also apply a rotation direction, together with the color as a two-pronged approach to distinguish between up-to-date and outdated libraries. The up-to-date ones,

[6]https://github.com/raphamorim/ranza



(a) *Ranza* overview, showing all 6 packages in use, 5 flagged as outdated (color and rotation) are candidates for an update.



(b) *babel-core'* coexisting libraries. The 5 that orbit around *babel-core*, show they are coexisting (have a *cc* with *babel-core*).

Fig. 2: *Ranza* Sol Mantra - showing 5 update opportunities based on their color and rotation (not shown) and the correlating update complexity based on their coexistence.

always orbit in a clockwise direction, while the outdated ones revolve in an opposite fashion. Because we are unable to show the rotation in this paper, the effect can be observed on our tool.

- ***Moons*** - represent coexisting libraries as orbiting the planet they coexist with[7]. The moons are duplicates to the libraries that the system uses with the purpose of showing which libraries should be considered to be updated together. As shown in Table I, *babel-core* has 5 coexisting libraries within the system, which is illustrated onto the Sol Mantra by adding these 5 as the orbiting moons, i.e. Fig. 2b.
- ***Size and Speed*** - similar to the two-pronged approach for the outdated flag, we provide a second visualization element that will show potential warnings for update opportunities. The size and speed, based on the number of moons (coexisting libraries), serve the purpose to concretely depict libraries with a high coexisting count. The higher the number, the larger the planet becomes and the slower it rotates on its orbits and the opposite, small planets that orbit faster, means that they have few or no coexisting libraries. For example, a planet with a large number libraries will be larger compared to one without. Oppositely, the speed will be considerably slower.

[7]Only planets can have moons

III. Application Examples

In this section we will demonstrate the applications of our tool through two distinct examples from the JavaScript npm ecosystem.

A. Target Data

To help us evaluate our tool, we selected and applied it to the node package manager (npm)[8]. It is the largest JavaScript repository, hosting over 230,000 packages with new ones constantly being added [10]. A package is treated as the system or libraries and it generally uses other libraries in the ecosystem. We selected the top 30 most used and liked packages for 2016. We then gathered various relevant data, such as stars, pull requests, issues, commits. contributors, releases, branches, dependencies, dependents in order to help us assess those projects.

In Table II, we provide a general statistics of our generated data and the filtering data. Most of the projects we use have a high star rate on *github*, with a maximum value of almost 70,000, mean and median with very close values. So we can safely state that these projects are popular within the software community. All of the projects are active and still under development. Proof can be found in the massive numbers of commits, pull requests, issues and releases. The average number of dependencies is at 9, but we were able to run a test on a package with close to 50 dependencies. Some packages did not use any libraries in their software, so we were unable to test them, hence, we were forced to discard 7 from our initial 30, resulting in 23 packages to test.

We implemented a script based tool to gather and collect the data needed for the visualization. In detail, to generate our data and calculate *cc*, we used packages available from the npm repository. In order to check if a library is outdated, we relied upon *is-outdated* package[9] and compared the result with the current version in the package.json file. For the coexistence coefficient, we used several packages in unison. We started by generating the users for every package, with *get-dependencies* package[10]. Secondly, we created the sets using *js-combinatorics*[11]. Finally, by using *comparray*[12], we

[8]https://www.npmjs.com/
[9]https://github.com/rogeriopvl/is-outdated
[10]https://github.com/SharonGrossman/get-dependencies
[11]https://github.com/dankogai/js-combinatorics
[12]https://github.com/JonathanPrince/comparray

TABLE II: Collected Data Summary

Data	Stars	Pull Requests	Issues	Commits	Contributors	Releases	Branches	Dependencies	Dependents
Minimum	917	0	0	186	15	11	1	0	1
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

extracted the common packages between all users of each set and calculated the final *cc* score.

To illustrate the use of our visualization and how a developer would use our tool, we selected two packages from our final 23. The first one is *react*[13], and is the highest starred package from our test data, with 67,706 stars on GitHub. For the second example, we chose *cheerio*[14] in order to showcase how our tool performs on a larger system. Both examples' data can be seen in Table III.

B. Example - react

React is a popular library used for interface creation. After generating the data, we can see the results, plotted onto our orbital layout. Fig. 3a Shows *react* system overview with a total of 5 dependencies, represented by the 5 planets orbiting the core. Continuing, 2 of them are green colored, i.e. up-to-date, and 3 are red, i.e. outdated.

From the overview, we can see that *loose-envify*, *object-assign* and *fbjs* are outdated and present an update opportunity. To evaluate the potential risks, their coexisting libraries have to be examined. By hovering over with the mouse and inspecting *loose-envify*, we can see that it has 100% *cc* with *prop-types* and *object-assign* (Fig. 3b).

fbjs has 3 coexisting packages in this example and as seen on Fig. 3c - 95.7% *cc* with *prop-types* and *object-assign*, and 23.16% with *loose-envify*.

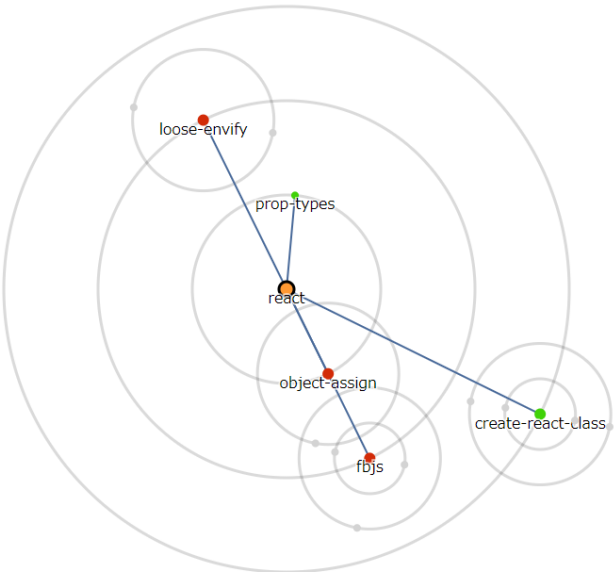
For *react* or tool results conclude that there are total of 3 update opportunities based on outdated flags. By inspecting them, the software developer can evaluate the update complexity with the reported *cc* values. *Loose-envify* has 100% *cc* with both of its moons, therefore careful evaluation should be made before updating. The case is similar for *fbjs*, since it has 95.7% with 2 of the moons. Although the *cc* with *loose-envify* is only 23.16%, it should still be evaluated.

C. Example - cheerio

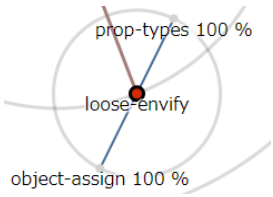
For the second example, we selected *cheerio* library package, providing implementation of core jQuery[15], designed specifically for servers.

Cheerio compared to the previous example, references more library packages in its code, and we will use that to show how our tool handles a larger system. Fig. 4, shows *cheerio's* SoL Mantra with 15 libraries that have high coexistence between each other.

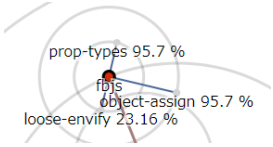
[13]https://github.com/facebook/react
[14]https://github.com/cheeriojs/cheerio
[15]https://jquery.com/



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



(b) *loose-envify* update opportunity complexity



(c) *fbjs* update opportunity complexity

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

After assessing the up-to-date libraries, we direct our attention towards the outdated ones. There are 11 update opportunities in total and 6 of them have high coexistence count - *istanbul*, *coveralls*, *parse5*, *htmlparser2*, *benchmark* and *lodash*. The full *cc* information can be observed on our

TABLE III: Examples Data

Data	Stars	Pull Requests	Issues	Commits	Contributors	Releases	Branches	Dependencies	Dependents
react	67,706	125	599	8640	1011	61	38	4	12,479
cheerio	12,298	16	132	1043	88	52	14	18	5358

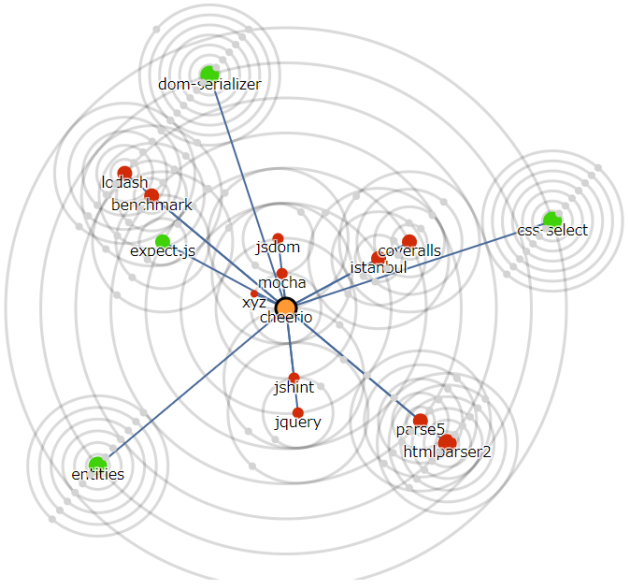


Fig. 4: *Cheerio* SoL Mantra - 11 outdated and 4 up-to-date libraries.

tool's result page.

Lastly in this example, we observe an interesting case, where *xyz* package does not have any coexistence on its own, but coexists with the others, apart from *dom-serializer*, with varying coexistence - from 0.05% with *lodash*, to 48% with *dom-serializer*.

From the *cheerio* overview visualization, we see that it uses packages which have high coexistence amongst each other. All the 11 outdated flags should carefully be inspected to evaluate the update complexity of each individually.

In conclusion, in both examples, our tool flagged the outdated libraries. Even in a bigger system like *cheerio* all 11 outdated libraries are easily detectable. Through *cc* we evaluate the complexity of each update opportunity. In smaller systems like *react*, the libraries didn't have high coexistence count but have high *cc* values, thus an easy update decision could be made. For the bigger system case where most packages have a high coexistent count, each update opportunity is more intricate and must be carefully evaluated.

IV. Threats to Validity and Limitations

Our collected data is based on established projects that have been developed by dedicated teams, hence, we have not tested smaller or personal projects. The projects we tested, could have specific library version for internal reasons. Further evaluating with use cases by real-life developers is required.

Because we only evaluate the data provided by the npm repository, we do not consider if the library references within the code are being used. Our tool only lists those that are registered and written on the *package.json* file of the respective project.

Improvements to SoL Mantra could be necessary based on use cases feedback. First and foremost, for larger software systems that use library packages with long names, the ones placed near the core, will become unreadable, unless focused specifically. Second, if all libraries have data similarity, they could overlap in neighboring orbits, which reduces visibility.

V. Conclusion and Future Work

In this paper, we propose our Sol Mantra tool, providing information for library update opportunities to software maintainers, by applying the coexistence coefficient.

Future work includes evaluating the usefulness, effectiveness and scalability of the SoL Mantra, by creating test cases, and including feedback from software maintainers. Depending on the results, further adjustments and enhancements will be added. Another potential visual element to explore for the future is the distance from the core. Currently, it serves only an aesthetic function to increase visibility, i.e. planets with higher moon count are pushed further out.

Acknowledgments

This work is supported by JSPS KANENHI (Grant Numbers JP25220003).

References

[1] C. Ebert, "Open source software in industry" in IEEE Software, Vol. 25, No.03, pp. 52-53, 2008.
[2] R. G. Kula, D. M. German, A. Ouni, T. Ishio and K. Inoue, "Do developers update their library dependencies?", Emp. Soft. Eng., 2017.
[3] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto and S. Panichella, "How the Apache community upgrades dependencies: an evolutionary study", Emp. Soft. Eng., Vol. 20, No. 5, pp. 1275-1317, 2015.
[4] C. Bogart, C. Kästner and J. Herbsleb, "When it breaks, it breaks: how ecosystem developers reason about the stability of dependencies", ASE (Workshop SCGSE), pp. 86-89, Lincoln, NE, 2015.
[5] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic Versioning versus Breaking Changes: A Study of the Maven Repository", 14th IEEE SCAM, pp. 215-224, 2014.
[6] R. G. Kula, D. M. German, T. Ishio, and K. Inoue, "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.
[7] R. G. Kula, C. De Roover, D. M. German, T. Ishio and K. Inoue (2014), "Visualizing the Evolution of Systems and Their Library Dependencies" in 2nd IEEE VISSOFT, pp. 127-136, Victoria, BC, Canada, 2014.
[8] Y. Yano, R. G. Kula, T. Ishio and K. Inoue (2015), "VerXCombo: An interactive data visualization of popular library version combinations" in 23rd IEEE ICPC, pp. 291-294, Florence, Italy, 2015.
[9] R. G. Kula, C. De Roover, D. M. German, T. Ishio and K. Inoue, "Modeling Library Popularity within a Software Ecosystem", Tech. Rep. Osaka University, Software Engineering Laboratory.
[10] E. Wittern, P. Suter, S. Rajagopalan, "A Look at the Dynamics of the JavaScript Package Ecosystem", 13th MSR, pp. 351-361, Austin, TX, USA, 2016.