Visualizing the Evolution of Systems and their Library Dependencies

Raula Gaikovina Kula*, Coen De Roover*†, Daniel German*‡, Takashi Ishio*, Katsuro Inoue*
* Osaka University, Osaka, Japan † Vrije Universiteit Brussel, Brussels, Belgium
‡ University of Victoria, Canada
Email: {raula, coen, cderoove, ishio, inoue}@ist.osaka-u.ac.jp
dmg@uvic.ca

***Abstract*—System maintainers face several challenges stemming from a system and its library dependencies evolving separately. Novice maintainers may lack the historical knowledge required to efficiently manage an inherited system. While some libraries are regularly updated, some systems keep a dependency on older versions. On the other hand, maintainers may be unaware that other systems have settled on a different version of a library. In this paper, we visualize how the dependency relation between a system and its dependencies evolves from two perspectives. Our system-centric dependency plots (SDP) visualize the successive library versions a system depends on over time. The radial layout and heat-map metaphor provide visual clues about the change in dependencies along the system's release history. From this perspective, maintainers can navigate to a library-centric dependants diffusion plot (LDP). The LDP is a time-series visualization that shows the diffusion of users across the different versions of a library. We demonstrate on real-world systems how maintainers can benefit from our visualizations through four case scenarios.**

I. Introduction

Dependence on third-party software libraries has become standard practice in both open source and industrial software engineering [1], with a vast source of libraries from large repositories such as SourceForge[1] and Maven Central[2]. Systems now rely on several dependencies of different libraries such as ASM[3], GOOGLE-GUAVA[4], JUNIT[5] and popular frameworks like SPRING [6] and HIBERNATE [7]. As these libraries each evolve independently from the system and from each other, tracking their evolution becomes important for the maintainers of a system.

As part of software maintenance, upgrading (or updating which we will use interchangeably) to a newer version of an outdated library may seem an obvious decision with advantages such as patched vulnerabilities, access to new features and continued support. However, deciding whether to upgrade requires careful consideration for systems with complex dependencies. For instance, knowledge of which dependencies were adopted at the same time may indicate relevance. Maintainers then can use this information to trace and assess respective affected system structures. Knowledge about a system's past upgrade decisions with respect to a library can help maintainers. Examples include significant dependency changes such as dropped and adopted libraries. Such historical information is particularly useful for novice maintainers and maintainers of poorly documented systems with many dependencies.

More seasoned maintainers, on the other hand, can benefit from knowledge about upgrade decisions made by different systems. Examples include identifying opportunities for upgrading to a newer version of a library as well as opportunities for migrating to a different library altogether. For instance, many systems might settle for a particular version because the next one has introduced many breaking API changes. Recognizing migration opportunities requires considering the dependency decisions of systems with similar dependencies. Many systems might abandon a particular library in favour of an equivalent one that is more frequently maintained or has better documentation.

In this paper, we visualize the evolution of systems and their library dependencies from two perspectives. Our *System-centric Dependency Plot* (SDP) provides an intuitive overview of the evolution of the dependencies of a system as it evolves. Different types of dependency changes can be discerned easily. Maintainers can differentiate between dependencies that are regularly updated and those that do not change. We use a *heat-map* metaphor to characterize the willingness of a system to adopt newer versions of a library as they are released.

From within the SDP, users can access library-specific usage and diffusion information by selecting a single dependency. The *Library-centric dependents Diffusion Plots* (LDP) that is shown to this end incorporates the "wisdom-of-the-crowd" by analyzing how other systems use a library. LDPs visualize the diffusion of dependent systems between the different versions of a library as well as movement of systems between each version.

We demonstrate the usefulness of both visualizations in four maintenance scenarios. In addition, we discuss interesting visual observations in visualizations of real-world systems and libraries. We provide the following two contributions:

- We present a visualization to explain the current state of a software system using important dependency changes

[1]http://sourceforge.net/
[2]http://mvnrepository.com/
[3]http://asm.ow2.org/
[4]https://code.google.com/p/guava-libraries/
[5]http://junit.org/
[6]https://spring.io/
[7]http://hibernate.org/

in its history.
- We present a visualization to understand the 'diffusion' usage across different library versions.

II. Background and Related Work

This section is divided into two parts. In the first part, we detail existing work on the software evolution visualizations. In part two, we explain the grounding theory for each visualization and employed visualization techniques.

A. Software Evolution Visualizations

There exists a large body of work dedicated to the visualization of software; its structure [2], the different relationships and metrics [3], and at different levels of granularity [4] . In this section, we focus on software evolution visualizations.

The added *time dimension* makes visualizing the evolution of software systems much more difficult compared to visualizing a system snapshot. Studies have shown that evolution visualizations help recognize important changes in the software such as re-factorings and newly introduced modules [5], [6] . Most methods either display a sequence of static snapshots of the system or display its entire evolution in one image.

Researchers have used visualization techniques for software evolution at the source code, class and architecture levels. *Code Flow* [7] uses the cable-and-plug wiring metaphor to describe line-level code changes. Wettel and Lanza [8] proposed *TimeLine* to view at class level changes. *Hierarchical Edge Bundles* [9] visualizes the software architecture in terms of organizational changes. The *Evolution Matrix* [10], *RelVis* [11] and *City/Cities* [12] all present the evolution of different metrics associated with the software architecture.

In contrast, we present a *new* approach to visualize the evolution of third-party dependencies at the architectural level. The visualizations provide usage information, popularity and movements of dependencies between library versions.

B. Visualization Techniques and Representations

Our visualizations are inspired by a combination of real-world metaphors for graphical representation. Since software is virtual, the combination of metaphors and graphing techniques in a familiar context helps users to immediately conceptualize a system and its dependent library relationships.

System-centric. Our system centric perspective is inspired by *Dendrochronology* [13], the scientific method of dating based on the analysis of patterns of growth rings. Starting from the center, each ring signifies a system release version. Similarly, RelVis [11] employs *Kiviat diagrams*. Other radial type visualizations are Sunburst [14] and Hive Plots [15]. Key benefits of using a radial layout is added structure for easier detection of patterns.

Heat-maps are useful to highlight intensity or importance of a particular phenomenon. Of particular interest was how Benomar [16] used heatmaps for easier understanding. Combining the radial design with the use of heat-map colours, we are able visualize the evolution of library dependencies and relative usage at that point in time.

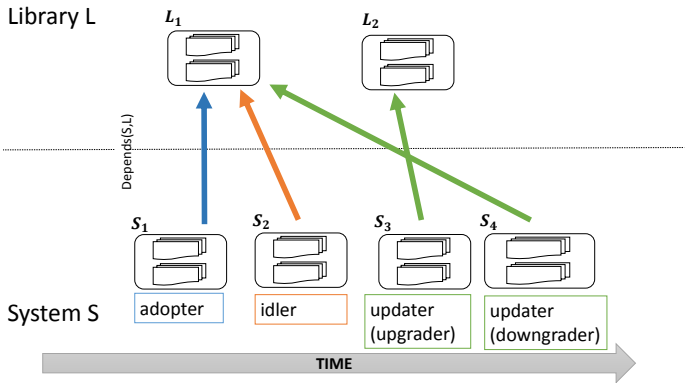


Fig. 1: Dependency relations between system versions and library versions.

Library-centric. Our library-centric perspective is inspired by the Diffusion of Innovations Theory [17], which implies a process of which new ideas/concepts are spread through the population. First we start with a slow adoption. Then there is a steady rate of adoption, then finally adoption slows down after reaching a critical mass. The 'S-Shape' growth curve (sigmoid curve), which can be expressed as a mathematical model, is commonplace in many fields of biology, medicine, economics and the social sciences [18]. It can describe and predict the evolution/growth of a quantitative measure over time. In software engineering, this curve has been applied primarily in the context of software reliability (e.g., [19], [20], [21]).

Work closely related to our library centric view is the usage 'popularity' of software libraries [22]. Mileva et al. study popularity over time to identify the most commonly used library versions [23]. Other work such as De Roover et al. explored library popularity in terms of source-level usage patterns [24]. In our visualizations, we express popularity as an aggregation and also track movements between different library versions.

III. System and Library Dependencies

In this section, we first introduce the necessary terminology and model for reasoning about dependencies between evolving systems and libraries. All examples are based on Figure 1.

- **Versions.** We refer to both system and library releases as versions. Conventions of releases are usually project specific. In this example, system S has 4 versions $S_1, .., S_4$ and L has L_1 and L_2 two respectively.
- **Dependency Relations.** A dependency relation is when a system starts using a library as its dependent. In this example, we have two versions of the library L_1 and L_2 and four versions of a system $S_1, .., S_4$. System versions S_1, S_2, S_4 depend on L_1 while system version S_3 depends on L_2.
- **Dependency Relations Change Types.** We classify system versions in terms of a change in the dependency relations. An *adopter* system version starts using a library

for the first time (i.e., it has not used previous versions). In our example in Figure 1, S_1 is an adopter of L_1.
An *idler* is a system version that depends upon the same library version as its immediate predecessor. Hence, in our example S changes from being an adopter in its version (S_1) to be an idler in version 2 (S_2).
An *updater* is a system version of which the previous version depended upon a different library version. In Figure 1, both S_3 and S_4 are updaters (for S_3 the dependency relations changed from (S_2, L_1) to (S_3, L_2). Note that an updater is either an *upgrader* or a *downgrader*. The former update to a newer version of the library, while the latter revert to an older. S_3 is an upgrader, while S_4 is a downgrader.
Finally, *dropper* is a system version of which the current version ceases the dependency relationship. Note that a *dropper* can revert to an *adopter* of a different library version or resume being an *idler* of a previous version.

IV. Visualization Design

Our main objective is to provide intuitive visual clues to assist maintainers with library upgrade decisions. Our visualizations are implemented through two separate, but linked perspectives. In this section we discuss three important visual aspects:

- *Layout/Metaphor Design* gives the different perspectives. Aids organization and structure of the dependency relations.
- *Shape Design* for the data points used to differentiate dependency relations types.
- *Choice of colour schemes and connecting lines* are used to differentiate between different versioning of systems and libraries.

In the next two subsections, we describe a) system-centric (System Dependency Plots) and b) library-centric (Library Diffusion Plots) using the three aspects. Users can easily navigate between perspectives by clicking on the respective points of the visualization.

For our examples, the *dependency relations* are derived from the dynamic linking of libraries (pom.xml [8]) provided from the Maven 2 Central Repository [9] ecosystem.

A. System-centric Perspective

System-centric Dependency Plots (SDP) provide a chronology of the evolution of the library version dependencies over all system versions. We use the example in Figure 2 to illustrate the different aspects of a SDP. The data used is 12 versions of the FindBugs system (ver. 0.9.4 - 2.0.1), plotting 192 dependency relations across 16 different libraries.

Depicted in Figure 2(a), each axis in a SDP represents a library that it depends upon. Starting from the center, each 'ring' represents a released version of the subject system. Time between releases is represented by the distance between rings.

[8]http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

[9]http://mvnrepository.com/

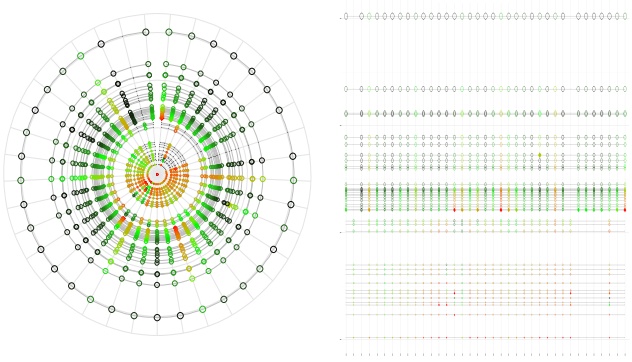
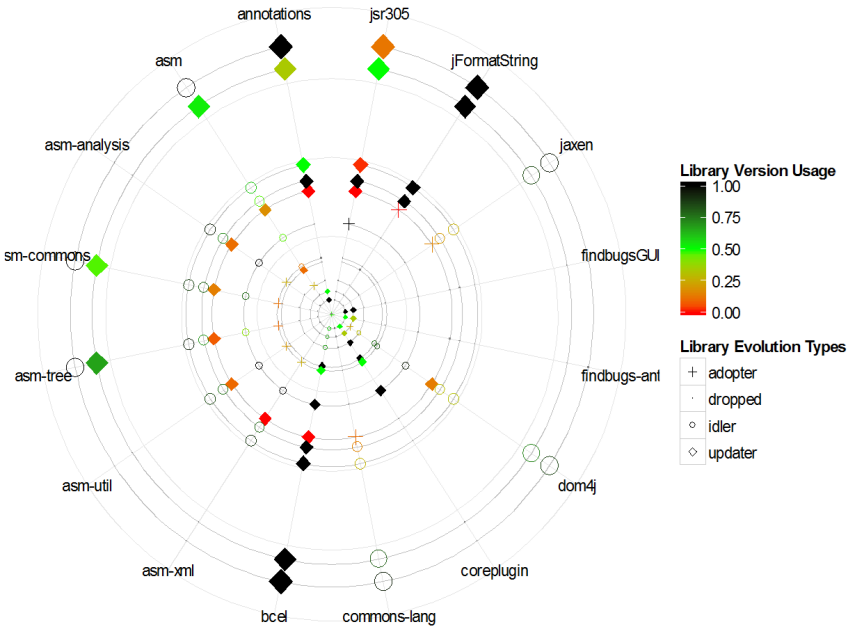


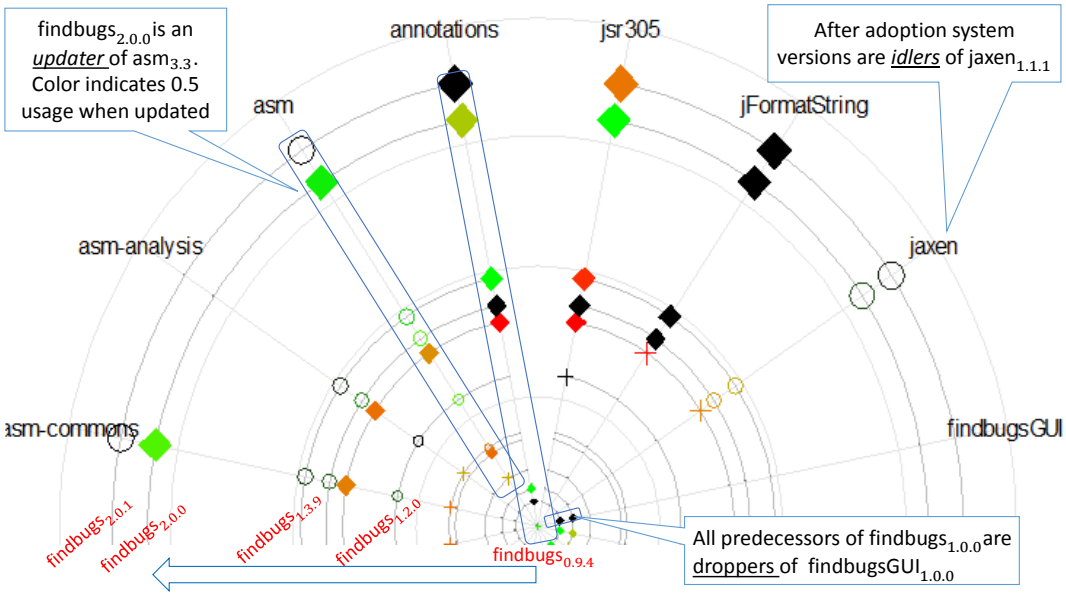
Fig. 3: Radial vs. linear layout. An increase of libraries on the x-axis inadvertently increases the size of the linear layout.

On each ring, each of the dependencies used by that release is denoted. The shape and color of the dependency represent the type of dependency relationship and the version. Next, we describe in detail based on the three visual aspects.

- **Layout/Metaphor Design.** Dependency relations are plotted using a radial coordinate system, with the axes representing a dependent library. As depicted in Figure 3, the radial design manages to keep structure as the number of libraries increases. The two-dimensional radial design provides a more organized view of the 'big picture' between all dependencies. Moreover, hive plots [15] and sunburst diagrams [14] argue radial represent a more rational visualization as opposed to the conventional edge-and-node network diagrams.
  The distance from the center corresponds to time since the first system release. As seen in Figure 2(a), the distance between the rings indicate the time between system releases. The angle and relative position of dependencies does not carry information. The major drawback to our design decision is the minimized visual space of older system versions.
- **Shape Design.** On each ring, the shape of the points describes the system's dependency relation to that particular version. $+$ (*adopter*) indicates when the system has first adopted the library. $\diamond$ (*updater*) indicates that the system is an *updater* to a new library version. $\circ$ (*idler*) indicates an existing dependency is maintained. Additionally, points that are not longer plotted (from the inside out) indicate the system is a *dropper* of that library. To reduce cluttering and overlaying of shapes, the outlines of the shapes were favored as opposed to fills. The only exception is for *updater* shapes, to highlight when a library has been upgraded. For aesthetic purposes, the shapes are sized proportionally to its distance from the center
  The shapes intuitively provide information on the regularity when the system updates its libraries. An example of this is shown in Figure 2(b). The appearance of updaters in consecutive system versions indicate that the FindBug



(a) Overview. According to the library version usage scale, the red indicates 'low usage', green 'mid usage' and black 'high usage' at that point of dependency. Figure 2(b) zooms in on the top-half of the plot



(b) This is a zoomed in cross-section of Figure 2(a). Note that *dropped* libraries are not plotted.

Fig. 2: SDP for FINDBUGS system. (a) gives an overall view and (b) is a zoomed in cross-section of with specific libraries on the axis. Starting from the center, each ring represents a system release. The relative distance between rings is the time between releases (weeks)

system regulary updates its libraries.

- **Colour and line schemes.** We assume most developers are less inclined to updated lesser used library versions. A heat-map colour intensity schematic can be used to provide visual hints about the system's willingness to adopt *newer* library versions. According to the Diffusion of Innovations, systems of newer (low usage) library versions are known as *early adopters*.
  Using a library version's usage, we express the *willingness* as a ratio of the usage when it was adopted over the current usage within a repository of projects. For example using the Maven repository, in Figure 2(b), when $FindBugs_{2.0.0}$ was released, $asm_{3.1}$ was updated to $asm_{3.3}$. At this point, there was 36 similar systems depending on $asm_{3.3}$. Currently 78 systems use $asm_{3.3}$, thus, resulting to a usage ratio of 0.52. (green intensity). We can also infer that FINDBUGS system is willing to update to *low usage* versions of the library JSR305 (red colour).
  Formally, we define time as t and $usage$ represents the ratio of systems that are using the library. Suppose vt is the time when a system is released. Therefore, $usage_{vt}$ represents the usage at the point in the evolution. Taking the current time as ct as a reference, we can identify its relative usage as a ratio:

$$Usage = \frac{usage_{vt}}{usage_{ct}} \tag{1}$$

  We can see in the example in Figure 2(a), the continuous colour scale depicts different intensities of library version usage. Relatively at that point of dependency, red indicates 'early adopter systems that there is a higher risk'. Likewise, green indicates that 'adopter systems to an attractive library with the rest of the majority '. Finally, black indicates 'adopter systems to an already stable library with other laggard systems'.

B. Library-centric perspective

The heat-map aspect of the SDP introduced in the previous subsection provides a subtle hint of the usage properties of a library version. Clicking on any library version will navigate the user to the library-centric diffusion plot. This section discusses this perspective in detail.

From a library-centric perspective, systems are viewed as dependents of a library. We define the *diffusion* of a library as the extent to which—and at what rate—its individual versions are spread among its dependents over time. We introduce *Library-centric dependents Diffusion Plots (LDP)* as a means to visualize diffusion over time

We use Figure 4 to illustrate the aspects of the LDP. We use a time-series graph technique, aligning the aggregation of system versions on the y-axis. The aggregation is a cumulative sum of systems that have a dependency relations with a specific library version. For basic understanding, our example plots a single system's library usage. A typical LDP is more complex, including all systems with a dependency relation to a library.

- **Layout/Metaphor Design.** Based on the diffusion of innovation theory, we expect that plotting the number of systems that have adopted a given version of a library would result in an *s-shaped (sigmoid) growth curve*. The initial stage corresponds to a period in which it is beginning to be adopted (by early adopters). At some point it starts to grow rapidly. This phase is followed by a transitional one in which the library version still gains dependent system versions, but at a slower rate. Finally, the function experiences a plateau phase in which growth ceases and the amount of dependants remains constant.
  We use the aggregation of systems to describe this growth function. The time-series on the x-axes allows the user to visualize the popularity of library versions at any point in time. As opposed to usage trends, aggregation can be mathematically modelled as a *cumulative growth function*.
- **Shape Design.** Each point is plotted using a specific shape to indicate the dependency relations between that particular system and the library. $+$ indicates *adopter*, $\circ$ indicates *idler* and $\diamond$ indicates an *updater*. Shapes give an indication of the types of systems that have adopted a particular library system. For instance, it gives the user information if a newer library version is attracting adopters and updaters. The occurrences of many idlers may pertain to a more stable library version.
- **Colour and line schemes.** Colours are used to classify the different library versions. As seen in Figure 4, it is a visual aid for the growth curve.
  The connecting lines group versions of a system. In the example, since it a single system, all points are connected. Consequently, the lines track *system movement* between the different versions. The system movements indicate if systems are willing to upgrade or move away from a library version.

V. ILLUSTRATIVE CASE STUDIES

To illustrate the use of our visualizations, we designed a cognitive walk-through of how a developer would use our tool with four case scenarios. First we introduce the scenario setting. Then we breakdown the setting into four case scenarios.

A. Case Setting

Our visualizations are targeted towards maintainers of a system. Familiarity of system, poor documentation or system complexity are motivating factors. We use the following case scenario to illustrate the challenges of upgrading libraries:

> *'Rusty is a **new** maintainer to a software project. Rusty notices that some of the system's library dependencies are outdated. Simply upgrading to the latest versions of all dependencies seems natural, however, Rusty does not know where to start. How to help Rusty?'*

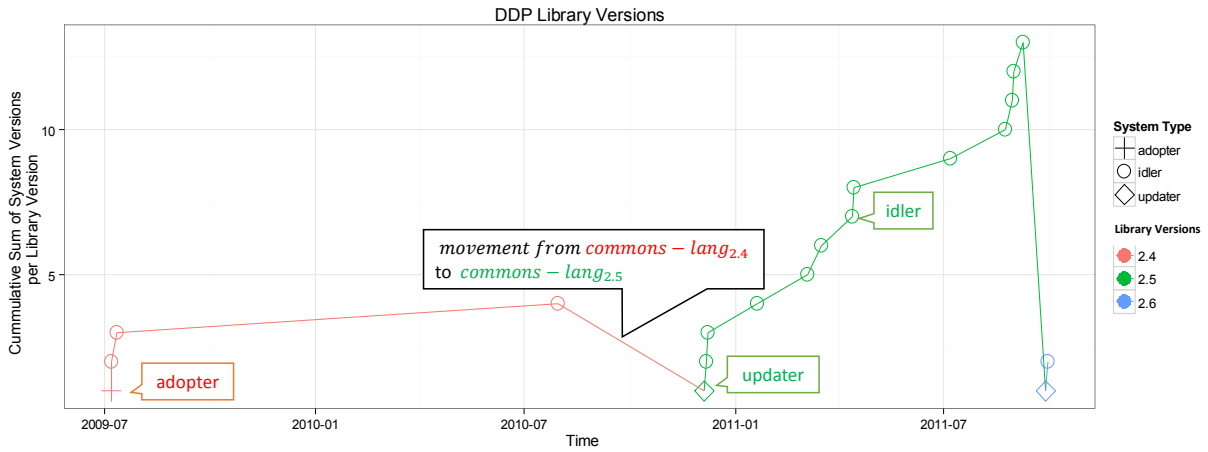


Fig. 4: Simplified example of a LDP with a single system for the COMMONS-LANG library. A typical LDP is composed of many systems.

To this end, Rusty is faced with a) understanding the systems dependency structure to prioritise which libraries should be upgraded and b) identifying suitable candidate library versions for upgrade.

Current state of the art tools and environments such as Eclipse [10] and its various plugins can allow Rusty to view the current snapshot of a system and respective library dependencies. However, most lack historic evolution information. The following two scenarios are related to how the system's dependency evolution history can be beneficial:

- **S1. Rusty wants to understand the regularity of system dependency changes.** The evolution history gives an indication of the frequency in relation to the version releases. It would also be useful if a system is more inclined to risk by adopting a newer version or is a either a regular updater or would rather wait until a library version is used by other similar systems before adopting.
- **S2. Rusty wants to understand what important structural dependency events have occurred.** Dependency relation changes such as dropped and adopted libraries can provide clues for important structural changes. Patterns can be used for understanding various historic events between dependencies.

In order to keep up-to-date with the library, Rusty also needs to know if the system libraries are in need of upgrades. Since Rusty is a typical wary developer, he wants to know what other similar systems that have reused the same libraries are doing. In this paper, we define a library versions *'attractiveness'* as highly favourable upgrade candidate. Based on the Innovation of Diffusion theory, we assume its successful diffusion into the population implies low risk with high benefit. The next two scenarios illustrate this viewpoint:

- **S3. Rusty wants to know the current 'attractiveness' of any library version.** Understanding the movement of adopters, idlers and updater systems provides visual clues on its 'attractiveness'.

[10]https://www.eclipse.org/

- **S4. Rusty wants to know if newer releases are viable candidates for updating.** Assessment of the 'attractiveness' of newer library versions can assist maintainers with the upgrade decisions.

B. Real World Examples

The first two scenarios stay in the system-centric perspective. The latter involve transition to the library-centric viewpoints.

For this study, we found that the FINDBUGS [11] system serves as a perfect example of our cognitive walk-through. FINDBUGS is a java program that uses static analysis to look for bugs in Java code.

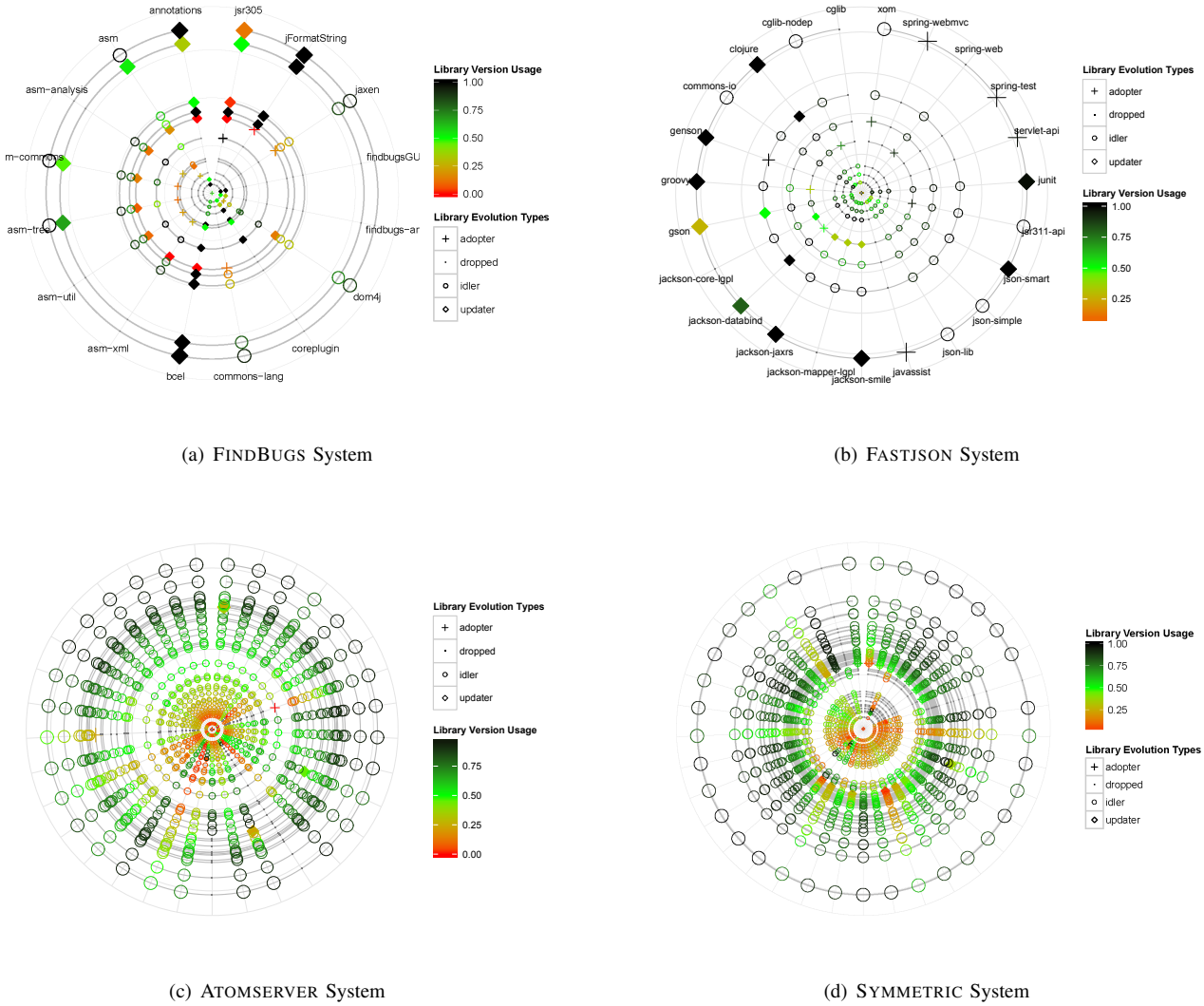
Additional examples are provided. FASTJSON [12] is a small and fast polymorphic JSON serializer. ATOMSERVER [13] is a generic data store implemented as a RESTful web service. Finally, SYMMETRICDS [14] is a open source software for both file and database synchronization for synchronization, and transformation across networks in a heterogeneous environment.

The *dependency relations* are derived from the dependency information provided from the Maven 2 Central Repository [15] ecosystem. Note that most projects in this repository are open-source Java, Scala or Clojure libraries. The visualization are generated from 188,951 dependency facts. Details of the data can be downloaded from our website [16].

C. Scenario Walk-through

> *'Rusty is a **new** maintainer to* FINDBUGS. *Rusty is able to view all dependencies but due to the complex nature of dependencies is hesitant on what dependencies are viable candidates for upgrading.'*

[11]http://findbugs.sourceforge.net/
[12]https://github.com/alibaba/fastjson
[13]http://atomserver.codehaus.org/
[14]http://www.symmetricds.org/
[15]http://mvnrepository.com/
[16]http://sel.ist.osaka-u.ac.jp/~raula-k/LibraryDiffusion2/index.html



(a) FINDBUGS System

(b) FASTJSON System

(c) ATOMSERVER System

(d) SYMMETRIC System

Fig. 5: Comparison of the SDP between the different systems

Using the Figure 2, we demonstrate how Rusty can use the SDP:

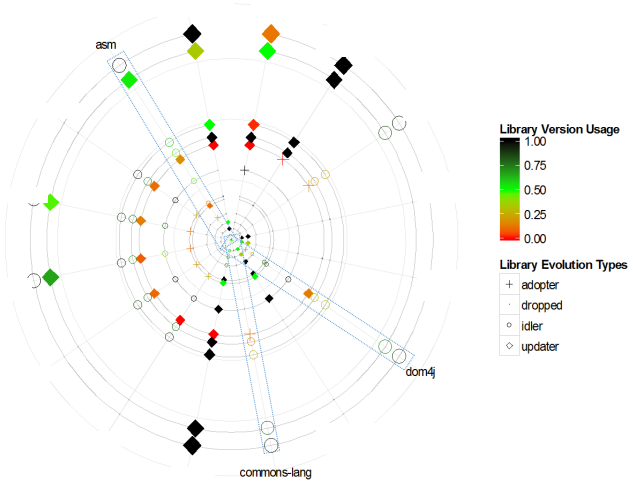
- ***S1. Rusty wants to understand the regularity of system dependency changes.*** Firstly, from a holistic view in Figure 2(a), Rusty can deduce alot of the dependencies were updated in a version just before the current version. Visually, Rusty can easily differentiate which dependencies were upgraded and the current usage. For instance, the gradient of the colours of the latest versions suggest that the current dependency relations have higher usage by peer systems.

Additionally, Figure 5 depicts the SDP of different systems. Intuitively, we can see that FINDBUGS and FASTJSON have more regular updating of their dependency relationships than ATOMSERVER and SYMMETRIC. Note that both ATOMSERVER and SYMMETRIC have ceased to upgrade their dependency relations, especially with the latest system versions.

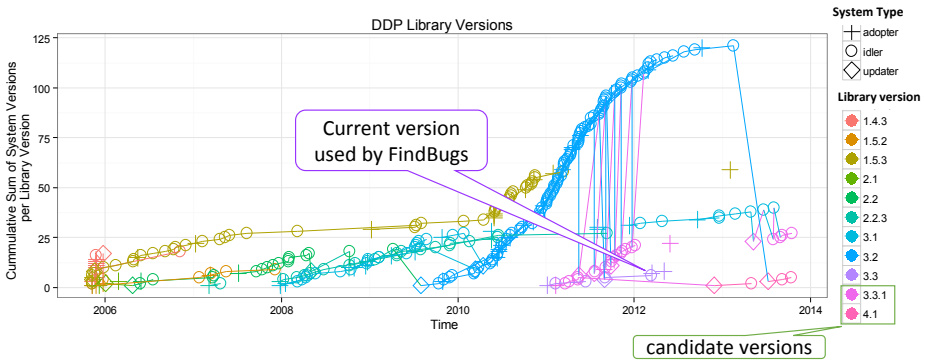
Therefore to address **S1**, *Rusty now understands the regularity of upgrade and decide which specific libraries are candidates for upgrading.*

- ***S2. Rusty wants to understand what important structural dependency events have occurred.*** First, Rusty looks for *dropped* dependency relations. Referring back to Figure 2(a), he notices that $asm - util$, $asm - xmil$ and $asm - analysis$ were dropped at the same time, thus hinting there may be an association between them. Additionally, FINDBUGS system specific *plugins* such as $findBugs - ant$, $findBugsGui$ and $coreplugin$ libraries are no longer depended upon.

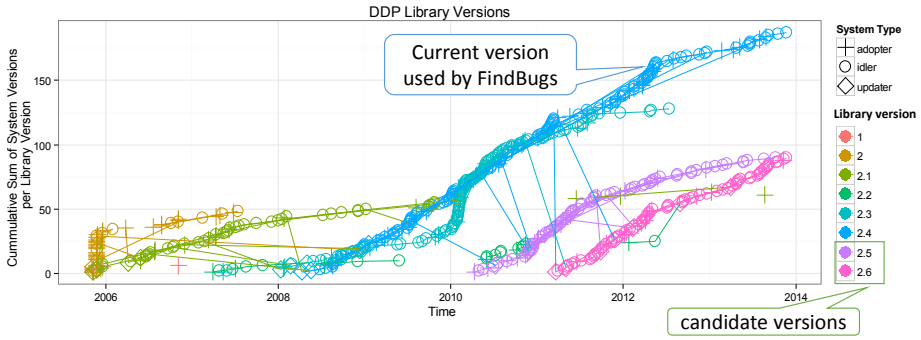
Therefore to address **S2**, *Rusty now understands the important changes and associated dependency relations.*



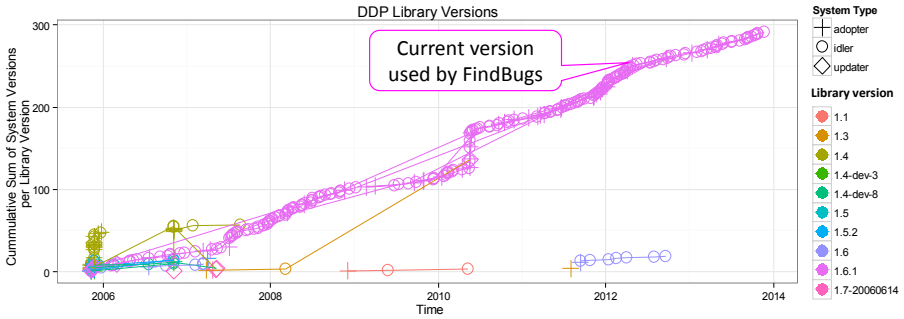
(a) SDP for FINDBUGS system



(b) LDP for ASM library



(c) LDP for COMMONS-LANG library



(d) LDP for DOM4J library

Fig. 6: Transition from SDP to LDP Visualizations

"Now that Rusty understands the current situation of the different library dependencies, he is looking for opportunities to update library dependencies."

From the SDP perspective, Rusty can now navigate to the library-centric LDP to understand the diffusion state of the library. We assume that Rusty is now considering updating ASM, COMMONS-LANG and DOM4J library versions.

Figure 6 illustrates the transition. By clicking on the axis of any of the highlighted libraries (Figure 6(a)), the user is taken to the library's respective LDPs (Figures 6(b), 6(c) and 6(d)). Now we can return to address **S3** and **S4**:

- ***S3. Rusty wants to know the current 'attractiveness' of any library version.***
  The latest version of FINDBUGS depends on is $asm_{3.3}$. Figure 6(b) suggests that this version is not very popular with similar systems. The curve of $asm_{3.1}$ suggests that it is the most popular version. The movement between $asm_{3.1}$ and $asm_{3.3}$ suggest development or testing reverts between versions. Since there is evidence of upgrading, Rusty should consider this library as upgradable. Similarly, Rusty gives the same 'thought process' for both COMMONS-LANG and DOM4J library.
  As depicted in Figure 6(c), FINDBUGS currently depends on $commons-lang_{2.4}$. There is some movement to different versions $commons-lang_{2.5}$ and $commons-lang_{2.6}$. For $commons-lang_{2.4}$, the aggregation is at a constant state. Finally, $dom4j_{1.6.1}$ is currently depended by FINDBUGS. From Figure 6(d), we note that this is the dominant version, which has a constant growth rate. All movement looks to be adoption into this version.

Visually from the shape of the growth curve in Figure 6, Rusty is provided clues on the current state of the libraries. Therefore in response to **S3**, *we illustrate that the LDP visualization enables Rusty to understand the current state of the library that the system is using.*

- **S4. Rusty wants to know if newer releases are viable candidates for updating.** The first thing to look for is the popular or dominant versions. In the case of ASM 6(b) for instance, newer versions $asm_{3.3.1}$ and $asm_{4.1}$ exist. Although popular, $asm_{3.1}$ seems to be ceasing growth. Rusty has the three candidates, that depending on his preference, can choose to adopt.
  Similar analysis can be done from the COMMONS.LANG and DOM4J libraries. From Figure 6(c), $commons-lang_{2.6}$ is a very strong candidate for updating. As shown in Figure 6(d), $dom4j_{1.6.1}$ seems the only stable version with no competitive candidate versions.

At this point, we demonstrate that in **S4**, *Rusty can visually identify viable candidates for updating.*

Using our visualizations, Rusty can now consider:

- *Library version $asm_{3.3.1}$ is a viable candidate for upgrading.*
- *Both $commons-lang_{2.5}$ and $commons-lang_{2.6}$ library versions are viable candidates for upgrading.*
- *Library version $dom4j_{1.6.1}$ is depicted as more 'stable', thus upgrading is not recommended.*

VI. DISCUSSION

A. Generality

While we demonstrate our approach on dynamic java library linkages within the Maven 2 ecosystem, our visualization techniques can be applied for any dependency relations. Relationships could be extended between classes or packages and it is language independent.

B. Visual Scalability

Due to the radial layout of the SDP, scalability can be achieved without compromise to structure. Shown in Figure 5(d), the most dependency-intensive system that we analyze is the SYMMETRIC system. It comprises of 44 system versions across 37 libraries, totalling to 1,628 dependency relationships.

For LDP visualization, more data provides a better estimation of the diffusion curve. Available at our website, the largest LDP would be the JUNIT library. [17]. We used 14 versions to plot 11,771 system versions.

To reduce clutter, the tool-tip can be utilized for labels and highlighting individual points on the graphs. This is an example of utilizing effective interface manipulation techniques such as zoom and pan, highlight and mouse trigger events. Future enhancements may include dynamic removing uninteresting points of the visualizations.

C. Ease of Use

We designed simplicity by using only two perspectives. Evaluation of this is not implicit in this study. For future work, we plan to carry out both controlled and practitioners survey evaluations.

D. Practicality

The decision to update libraries is reliant on several factors and depending on both system and maintainers preference, can be a case-by-case evaluation. Therefore, our system provides a means for which maintainers to make a more informed decision based on historic data rather than guesswork.

In order to maintain up-to-date, usage data needs to be periodically updated. Historic data can grow quickly, thus requiring higher space and computation costs. We envision that effective optimization and filtering techniques can be implemented to properly manage this data.

[17] http://sel.ist.osaka-u.ac.jp/~raula-k/LibraryDiffusion2/DDP/DDPJunit.pdf

E. Additional Scenarios

Due to space requirements, the paper only covers general case scenario of the tool. Additional scenarios could be the following:

- **Recursive SDP views.** This could be useful especially to view the deeper 'transitive' library dependency relationships.
- **Library frameworks.** SDP view to understand the evolution of a framework. Maintainers of frameworks may want to understand the 'transitive' library dependencies of a framework.
- **Related system SDPs.** From the LDP view, click on other similar systems that are currently using a candidate update library version. This would allow users to 'peek' into the SDP for other systems.

We would like to extend our visualizations to track movement of similar systems across different libraries. We believe that this will provide maintainers with alternative library options.

VII. Conclusion and Future Work

We present two new visualization approaches of systems evolution with their library dependencies. Our visualizations capture how dependency relations between a system and its dependencies evolves from two perspectives. Our system-centric dependency plots (SDP) visualize the successive library versions a system depends on over time in a radial layout. It provides visual clues about the change in dependencies along the system's release history. From this perspective, maintainers can navigate to a library-centric dependants diffusion plot (LDP). The LDP is a time-series visualization that shows the diffusion of users across the different versions of a library. We demonstrate on real-world systems how our visualizations can benefit maintainers through four case scenarios. Future enhancements include additional interactive and dynamic transitions between the two visualizations.

VIII. Acknowledgments

Thanks to Prof. Makoto Matsushita for his useful comments. This work is supported by JSPS KANENHI (Grant Numbers 25220003 and 26280021) and the "Osaka University Program for Promoting International Joint Research.".

References

[1] C. Ebert, "Open source software in industry," in *IEEE Software*, 2008, pp. 52–53.

[2] H. Kienle and H. Muller, "Requirements of software visualization tools: A literature survey," in *Proc. of Int. Workshop on Visualizing Software for Understanding and Analysis, (VISSOFT2007)*, June 2007, pp. 2–9.

[3] S. Bassil and R. Keller, "Software visualization tools: survey and analysis," in *Proc. of Int. Workshop on Program Comprehension (IWPC 2001)*, 2001, pp. 7–17.

[4] P. Caserta and O. Zendra, "Visualization of the static aspects of software: A survey," *Trans on Visualization and Computer Graphics*, vol. 17, no. 7, pp. 913–933, July 2011.

[5] S. Eick, T. Graves, A. Karr, A. Mockus, and P. Schuster, "Visualizing software changes," *Trans. on Software Engineering*, pp. 396–412, Apr 2002.

[6] A. Hindle, Z. M. Jiang, W. Koleilat, M. Godfrey, and R. Holt, "Yarn: Animating software evolution," in *Proc. of Int. Workshop on Visualizing Software for Understanding and Analysis, (VISSOFT2007).*, June 2007, pp. 129–136.

[7] F. Chevalier, D. Auber, and A. Telea, "Structural analysis and visualization of c++ code evolution using syntax trees," in *Int. Workshop on Principles of Software Evolution (IWEPSE2007)*, New York, NY, USA, 2007, pp. 90–97.

[8] R. Wettel and M. Lanza, "Visual exploration of large-scale system evolution," in *Proc. of Working Conf. on Reverse Engineering, (WCRE2008)*, 2008, pp. 219–228.

[9] D. Holten and J. J. van Wijk, "Visual comparison of hierarchically organized data," in *Proc. of the 10th Joint Eurographics / IEEE - VGTC Conf. on Visualization*, 2008, pp. 759–766.

[10] M. Lanza, "The evolution matrix: Recovering software evolution using software visualization techniques," in *Proc. of the Int. Workshop on Principles of Software Evolution, (IWPSE2001)*. ACM, 2001, pp. 37–42.

[11] M. Lanza and S. Ducasse, "Understanding software evolution using a combination of software visualization and software metrics," in *Proc. of Langages et ModÃles Ã Objects (LMO2002)*, 2002, pp. 135–149.

[12] M. Pinzger, H. Gall, M. Fischer, and M. Lanza, "Visualizing multiple evolution metrics," in *Proc. of ACM Symposium on Software Visualization (2005)*, 2005, pp. 67–75.

[13] S. Arno and K. Sneck, "A method for determining fire history in coniferous forests of the mountain west," in *USDA Forest Service General Technical Report*, May 1988, p. 28.

[14] K. Andrews and H. Heidegger, "Information slices: Visualising and exploring large hierarchies using cascading, semi-circular discs," in *Proc. of Infovis1998*, 1998.

[15] M. Krzywinski, I. Birol, S. J. Jones, and M. A. Marra, "Hive plots: A rational approach to visualizing networks," *Briefings in Bioinformatics*, 2011.

[16] O. B. Omar, H. A. Sahraoui, and P. Poulin, "Visualizing software dynamicities with heat maps," in *Proc. of VISSOFT2013*, 2013.

[17] E. M. Rogers, *Diffusion of innovations*, 5th ed. Free Press, 08.

[18] G. Annadurai, S. Rajesh Babu, and V. R. Srinivasamoorthy, "Development of mathematical models (logistic, gompertz and richards models) describing the growth pattern of pseudomonas putida (nicm 2174)," *Bioprocess Engineering*, vol. 23, pp. 607–612, 2000.

[19] V. Almering, M. van Genuchten, G. Cloudt, and P. Sonnemans, "Using software reliability growth models in practice," *Software, IEEE*, pp. 82–88, 2007.

[20] A. Goel, "Software reliability models: Assumptions, limitations, and applicability," *Trans. Software Engineering*, pp. 1411–1423, 1985.

[21] S. Yamada, M. Ohba, and S. Osaki, "S-shaped reliability growth modeling for software error detection," *Trans. Reliability*, pp. 475–484, 1983.

[22] Y. M. Mileva, V. Dallmeier, and A. Zeller, "Mining api popularity," in *Proc.of the International academic and industrial conference on Testing - practice and research techniques (TAIC PART2010)*, 2010, pp. 173–180.

[23] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining trends of library usage," in *Proc. Intl and ERCIM Principles of Soft. Evol. and Soft. Evol. Workshops (IWPSE-Evol '09)*. ACM, 2009, pp. 57–62.

[24] C. De Roover, R. Lämmel, and E. Pek, "Multi-dimensional exploration of api usage," in *Proc. of IEEE Intl. Conf. on Prog. Comp.(ICPC13)*, 2013.