Osaka University Technical Report TR20062017 manuscript No.
(will be inserted by the editor)

Modeling Library Popularity within a Software Ecosystem

Raula Gaikovina Kula · Coen De Roover · Daniel M. German · Takashi Ishio · Katsuro Inoue

Online: June 20, 2017

Abstract The popularity of third party libraries such as the Maven Central and the CRAN repositories is a testament to software reuse activities in both open-source and commercial projects alike. Studies have also highlighted the risks and dangers associated when developers keep outdated library dependencies in their applications. Intelligent mining of these massive software repositories (i.e., super repositories) could unlock valuable knowledge pertaining to such dependency-related decisions and reveal trends within the software ecosystem. In this paper, we propose the Software Universe Graph (SUG) Model as a structured abstraction of evolving software and their library dependencies over time. To demonstrate practically and usefulness of the SUG, we conduct an empirical study using 6,374 Maven artifacts and over 6,509 CRAN packages mined from their real-world ecosystems. Results show the Maven ecosystem as having a more conservative approach to dependency updating, while the CRAN ecosystem exhibits a more always updated approach to dependencies. Novel visualizations of SUG model operations such as library coexistence pairings and dependents diffusion uncover evidences to explain popularity, adoption and diffusion patterns within each software ecosystem.

Raula Gaikovina Kula, Takashi Ishio
Nara Institute of Science and Technology, Japan
E-mail: {raula-k, ishio}@is.naist.jp

Coen De Roover
Vrije Universiteit Brussel, Belgium
E-mail: cderoove@vub.ac.be

Daniel M. German
University of Victoria, Canada
E-mail: dmg@uvic.ca

Katsuro Inoue
Osaka University, Japan
E-mail: inoue@ist.osaka-u.ac.jp

1 Introduction

Reusing software by depending on libraries is now commonplace in both open source and commercial settings alike [1,2]. Software libraries come with the promise of being able to reuse quality implementations, thus preventing *'reinventions of the wheel'* and speeding up development. Examples of popular reuse libraries are the SPRING [3] web framework and the APACHE COMMONS [4] collection of utility functions. Contributing to the popularity of these and other libraries has certainly been the ease through which they can be accessed nowadays from ecosystems formed by a collection of *super repositories* such as Maven Central [5], R's CRAN [6], Sourceforge [7] and GitHub [8].

With new libraries and newer versions of existing libraries continuously being released, managing a system's library dependencies is a concern on its own. Improper dependency management can be fatal to any software project [9]. As outlined in related studies [10–12], dependency management includes making cost-benefit decisions related to keeping or updating dependencies on outdated libraries. Such decisions are influenced by whether or not security vulnerabilities have been patched and important features have been improved, but also by the amount of work required to accommodate changes in the API of a newer library version. Recent studies on library APIs have shown that a developer responding to a new library updates is slow and lagging [13–16].

Meta-data recorded within these ecosystems can provide system maintainers valuable "wisdom-of-the-crowd" insights into these dependency-related questions. Building on our work on visualizing the evolution of systems and their library dependencies [17], we introduce the Software Universe Graph (SUG) as a means to model the realities of popularity, adoption and diffusion within a super repository. Popularity refers to the usage of a library over time. Adoption refers to systems introducing a new library dependency. Diffusion, inspired by use-diffusion [18], is a measure of the spread of library versions over dependent systems. The abstract nature of our SUG enables generalizing and hence comparing these aspects across different types of super repositories.

To evaluate the SUG model, we report on a large-scale empirical study in which we construct SUGs for a large collection of Maven and CRAN super repositories. Our goal is to evaluate: (1) construct real-world SUG models to show the practical application and (2) through several case study examples demonstrate the SUG usefulness when address library dependency management issues. Our key contributions are:

- We introduce the fully formalized SUG model for representing super repositories in a generic manner, which lends itself to being mined for insights about popularity, adoption and diffusion.
- We define several metrics related to popularity, adoption and diffusion —all in terms of formal operations on a SUG model. We also introduce several SUG-based visualizations.
- In a large-scale study, we build SUG models for the very different realities of the Maven and of the CRAN super repository. We demonstrate that our visualizations intuitively provide valuable insights for dependency

management. The study results empirically depict Maven users as reluctant to update to newer library releases, with older library releases deemed 'usable' by the crowd. CRAN dependencies are more disciplined in this regard.

2 Background

Studying library usage in terms of absolute popularity is not a new concept. Holmes et al. appeal to popularity as the main indicator to identify libraries of interest [19]. Eisenberg et al. improve navigation through a library's structure using the popularity of its elements to scale their depiction [20]. De Roover et al. explored library popularity in terms of source-level usage patterns [21]. Popularity over time has received less attention. Mileva et al. study popularity over time to identify the most commonly used library versions [22]. Follow-up work applies the theory of diffusion to identify and predict version usage trends [23]. Similar to our diffusion work, Bloemen et al. [24] explored the diffusion of Gentoo packages. Using the economic bass model, they modeled the diffusion of gentoo packages over time.

Other related work include Teyton and colleagues use of *'library migration graphs'* to identify candidate library migrations [11]. The migration graphs offers a cyclic approach, much different to our incremental SUG model. Additionally, our study investigates coexistence and diffusion instead of migration and our case study involves two different super repositories. Instead of a single-dimensional analysis of popular library usage, we present an extensively formalized SUG model with popularity and variety metrics and additional complementary diffusion and coexistence plot visualizations. This provides for a much richer understanding of any significant phenomena in the evolution of libraries. Our generic model provides for generalization of popularity across different super repositories.

3 Software Universe Modeling

This paper is concerned with intelligent mining of a large collection of software repositories within an ecosystem, defined as *super* repositories. More specifically, we present an abstract model to understand and compare adoption, diffusion and popularity within its particular *universe*. We conjecture that useful information such as popularity is indicative of a library reliability, which is measured by significant usage within the ecosystem.

3.1 Software Universe Graph

We present the *Software Universe Graph* (SUG) as a structural abstraction of a super repository. Figure 1 will serve as an illustration of the different relationships within the graph. Let $G = (N, E)$ for a graph G. N is a set

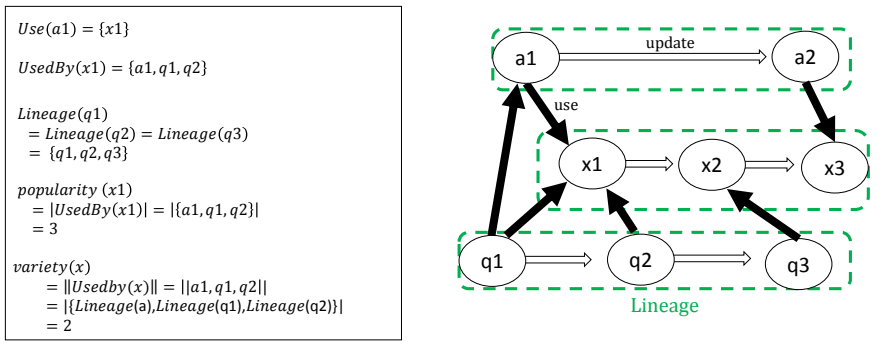


Fig. 1: A conceptual example of the Software Universe Graph with formalized definitions and notations.

of nodes, with each node representing a software unit. We consider both any system version, such as `SymmetricDs` version `3.6.3` ($SymmetricDs_{3.6.3}$) or a library version `Junit` version `4.11` ($Junit_{4.11}$) as software unit nodes. For any SUG, the edges E are composed of E_{use} and E_{update}. E_{use} is a set of *use-relations* and E_{update} is a set of *update-relations*. We first present E_{use} in Definition 1 and 2. E_{update} is then introduced in Definition 3.

Definition 1 *An edge $u \rightarrow v \in E_{use}$ means that u uses v. The defined functions of E_{use} are:*

$$Use(u) \equiv \{v | u \rightarrow v\} \tag{1}$$
$$UsedBy(u) \equiv \{v | v \rightarrow u\} \tag{2}$$

Use-relations can be extracted from either the source code or configuration files. As depicted in Figure 1, node $a1$ uses node $x1$. Also node $x1$ is used by nodes $a1$, $q1$ and $q2$. Parallel edges for node pairs are not allowed. In this paper, we focus on popular software units that are connected by many use-relation edges.

Definition 2 *For a given node u, popularity is the number of incoming use-relation edges and is defined as:*

$$popularity(u) \equiv |UsedBy(u)| \tag{3}$$

For instance in Figure 1, for node $x1$, $popularity(x1) = |UsedBy(x1)| = |\{a1, q1, q2\}| = 3$. As an extension, the popularity of any pair of nodes (u and v) is defined by the number of common nodes connected by an incoming edge. Formally,

$$popularity(u, v) \equiv |UsedBy(u) \cap UsedBy(v)| \tag{4}$$

We define u and v as being *coexistence pairs* if $popularity(u, v) \geq 1$. Take from Figure 1, $popularity(x1, a1) = |\{UsedBy(x1) \cap UsedBy(a1)\}| = |\{a1, q1, q2\} \cap \{q1\}| = |\{q1\}| = 1$. Therefore in the Figure, $x1$ and $q1$ are coexistence pairs.

Definition 3 *We represent an update-relation from node a to b using $a \Rightarrow b$, meaning that newer update b had been released from node a and is defined as:*

$$a \Rightarrow b \in E_{update} \tag{5}$$

Update-relations refers to when a succeeding release of a software unit is made available. Figure 1 shows that node $q1$ is first updated to node $q2$. Later on, node $q2$ is updated to the latest node $q3$. Hence, $q1 \Rightarrow q2 \Rightarrow q3$. We find that every node in the SUG should be denoted by three attributes: `<name,release,time>`. For a node u, we then define:

- **u.name** Name is the string representation identifier of a software unit. We introduce the name axiom: For nodes u and v, if $u \Rightarrow v$, then $u.name = v.name$ holds.
- **u.release**. Release refers the specific assigned change reference for a software unit. For nodes u and v, if $u \Rightarrow v$ then v is the immediate successor of u. Note that the versioning pattern may vary from project to project.
- **u.time**. Time refers to the time-stamp at which node u was released. For nodes u and v of $u \Rightarrow v$, $u.time < v.time$.

An example of the attributes can be shown with the JUNIT library. These attributes belong to the most recent release[1] (i.e., `<name = "junit", version= "4.11", time="2012-11-14">`). We define a set of nodes weakly connected by update-relations as a *lineage*. We are interested in all releases within a lineage.

Definition 4 *Lineage of a related set of nodes are determined through transitive update-relations. This is defined as:*

$$Lineage(u) \equiv \{v | v \overset{+}{\Rightarrow} u \ \vee u \overset{+}{\Rightarrow} v \ \vee u = v\} \tag{6}$$

where $a \overset{+}{\Rightarrow} b$ is the transitive closure on any update-relation $a \Rightarrow b$.

The name axiom proves that all names in a lineage are the same. A lineage of nodes is depicted in Figure 1, where $Lineage(q1) = Lineage(q2) = Lineage(q3) = \{q1, q2, q3\}$. The lineage function enables more dynamic operations. To differentiate between lineages, we now introduce an additional operator.

Definition 5 *We use the $\parallel S \parallel$ operator to represent the number of different lineage in a set of nodes in S.*

$$\parallel S \parallel \equiv |\{Lineage(u) | u \in S\}| \tag{7}$$

Looking back at the example in Figure 1, suppose $S = \{a1, a2, x1\}$. $\parallel S \parallel = |\{Lineage(a1), Lineage(a2), Lineage(x1)\}| = |\{\{a1, a2\}, \{x1\}\}| = 2$. Complex queries on our SUG model based on lineages are now possible. The previously defined *popularity* function alone is insufficient in reflecting the spread or diffusion of a software unit across the software universe. We introduce a *variety* function that allows use to measure diffusion.

[1] `http://mvnrepository.com/artifact/junit/junit/4.11`: accessed 2017-04-01

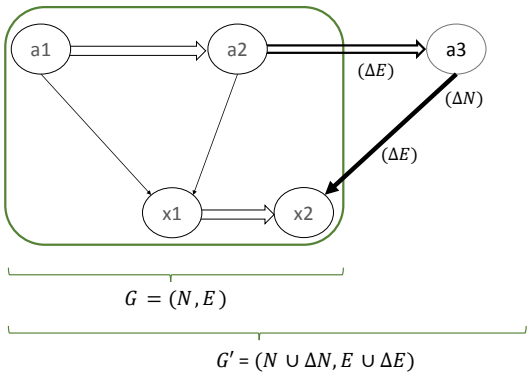


Fig. 2: Temporal property of the SUG

Definition 6 *Variety represents the number of different lineages that use a software unit.*

$$variety(u) \equiv \| \, UsedBy(u) \, \| \tag{8}$$

In Figure 1 we observe that node $x1$ is used by node related to $Lineage(a1)$ and $Lineage(q1)$. Hence, variety is 2. Formally, $variety(x1) = \| \, \{a1, q1, q2\} \, \| = |\{Lineage(a1), Lineage(q1), Lineage(q2)\}| = 2$.

Definition 7 *The SUG has temporal properties. This describes the simultaneity or ordering in reference to time. Let SUG $G = (N, E)$ be at time t. At time $t' > t$, we observe an extension of G, such that:*

$$G' = (N \cup \Delta N, E \cup \Delta E) \tag{9}$$

where $\Delta E \cap (N \times N) = \varnothing$.

Figure 2 illustrates the temporal properties of the SUG. Here, it is observed that G' is composed of G augmented with newly added node $a3$ and its corresponding $a3 \rightarrow x2$ and $a2 \Rightarrow a3$ relations. A SUG grows monotonically over time with only additions. Here we consider that modification or deletion changes on the SUG do not occur.

Definition 8 *A timed SUG specifies the state of the SUG at any point in time. So for a SUG $G = (N, E)$, we represent a timed SUG G_t at time t as a sub-graph of G. Formally,*

$$G_t \equiv (N_t, E_t) \tag{10}$$

where $N_t = \{u | u \in N, u.time \leq t\}$ and $E_t = \{e | e \in E \wedge e \in N_t\}$.

We are now able to describe the temporal properties of popularity. We introduce $Popularity_t(u)$ for a node u at time t. This provides the popularity of u for G_t[2].

[2] We define that $Popularity_t(u) = 0$ if $t < u.time$

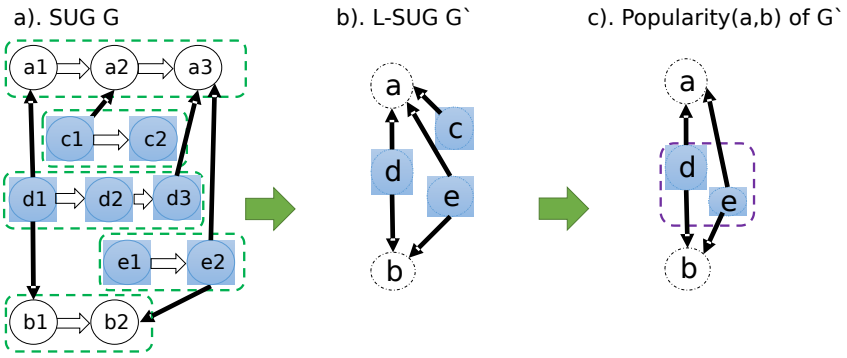


Fig. 3: Illustrative example of the coexistence pairing. Let Figure 3(a). shows a typical SUG G with respective lineages annotated. Figure 3(b) is the L-SUG $G\prime$. Finally in Figure 3(c), the popularity is determined. In this case the coexistence `popularity(a,b)` is 2.

3.2 Query Operations on the SUG Model

We utilize the SUG model to query and retrieve useful information from the software ecosystem. We now introduce *Coexistence Pairing* and *Diffusion Plots* as examples of the visualization of popularity. Our rational is that popular usage of a software unit is evident by successful adoption and diffusion over its predecessors. It is important to note that results do not indicate concrete evidence of library compatibility, instead popularity suggests some recommended usages based on the 'wisdom of the crowd'.

Coexistence pairing examine and explores occurrences of specific combinations between software components. We empirically visualize these coexistence pairs, deemed to be compatible by the 'crowd'. The coexistence popularity of two nodes (i.e., $popularity(u, v)$) is used to establish a pairing between the nodes. Alternatively, non coexistence pairs (i.e., $popularity(u, v) = 0$) suggest combinations that are possible but not seen yet. To aggregate nodes by lineage, we introduce a reduced SUG called Lineage SUG (L-SUG).

Definition 9 *A Lineage SUG is used to merge all related nodes into a single node. We define a Lineage SUG (L-SUG)* $G\prime = (N\prime, E\prime)$ *of an SUG* $G = (N, E = E_{use} \cup E_{update})$ *where:*

$$N\prime = \{Lineage(u) | u \in N\} \tag{11}$$

and

$$E\prime = \{Lineage(u) \rightarrow Lineage(v) | u, v \in N \wedge u \rightarrow v \in E_{use}\} \tag{12}$$

In the L-SUG, all nodes belonging to the same lineage are merged with all use-relations mapped onto the merged node. All update-relations are discarded. Almost all defined functions for the SUG like popularity, should the-

node	operation	coexistent rank for node a
s	popularity (a,s)	2
r	popularity (a,r)	1
y	popularity (a,y)	1
z	popularity (a,z)	1
b	popularity (a,b)	1

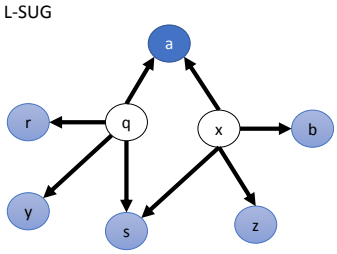


Fig. 4: Example of the coexistence operation. For the L-SUG $P\prime$, for a node a, the coexistence rank allows to determine popularity in relation to node a. In this example node s has the highest coexistence rank.

oretically apply on the L-SUG[3]. An example of the L-SUG is explained in Figure 3. Figure 3(a) shows a typical SUG G with respective lineages annotated. Figure 3(b) depicts the lineage SUG $G\prime$. In Figure 3(c), the L-SUG popularity of L-SUG nodes a and b is calculated. We now introduce the three different types of coexistence pairing operations as follows:

1. **L-SUG Lineage Pairs**. We use the *popularity* of two L-SUG nodes. Further exploration of the lineage pairs would lead to release pairs of these lineages.
2. **SUG Release Pairs**. Once interested lineage pairs are identified, maintainers next decide on popular release combinations. To this end, we use *popularity* of two nodes on the SUG to establish coexistence release pairs.
3. **L-SUG Lineage Coexistence Rank**. The coexistence rank returns popularity pairs relative to a specific L-SUG node. Formally, for a node u on a set of L-SUG $U\prime$, we compute all values of $popularity(u, v)$ for $v \in U\prime$ where $popularity(u, v) \geq 1$. We refer to them as the *coexistence rank*. For example in Figure 4, consider L-SUG node a. The coexistence rank returns 5 results, with node s having the highest coexistence rank of 2. Later in Section 5.2, we demonstrate how the coexistence rank may uncover intuitive insights.

As an extension on our work of Library Dependency Plots (LDP) [17], we introduce Diffusion Plots. For any software unit, DPs allow us to be able to plot and track both popularity and variety at any given point in time t such that $popularity_t(u)$ and $variety_t(u)$ for a node u. The DPs provide a temporal means to evaluate popularity and the adoptive behavior nature of a software unit within its universe.

We introduce the two main attributes of the DP:

– **Popularity and Variety Plots.** DPs plot both the $popularity_t$ and corresponding $variety_t$ on a SUG. We use the plots to understand the adoption and diffusion at both the lineage and release level.

[3] An exception would be L-SUG variety, which is a lineage within an lineage. We consider this to be out of scope for this study.

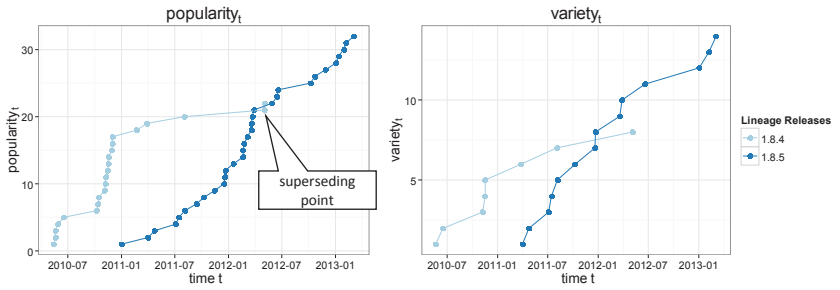


Fig. 5: A simple example of the diffusion plot for the Maven MOCKITO-CORE lineage release `1.8.4` and `1.8.5`

– **Diffusion Curve Types.** The DPs allow us to perceive patterns of popularity for a specific release belonging to a lineage. Additionally, the DPs provide insights on the adoption and diffusion within that universe.

In Figure 5, we show an example DP of the MOCKITO-CORE lineage from the Maven super repository. For illustration purposes –and to simplify the curve– this DP only shows two releases. Note the crossing of lines, which is described as the *superseding point* where MOCKITO-CORE$_{1.8.5}$ succeeds MOCKITO-CORE$_{1.8.4}$ in both *popularity*$_t$ and *variety*$_t$. Section 5.3 details the DP visualizations.

4 Empirical Study

Our aim of the empirical study is to demonstrate: (1) practical application of our formalized SUG model when applied to real-world data and (2) usefulness by providing insights and comparisons between two different ecosystems. In detail, we demonstrate how our model operations and visualizations supports describe real-world data. All tools, scripts, data and results are freely available from the paper's replication package at `http://goo.gl/cF2rJZ`.

To demonstrate the practicality of the SUG, we construct and apply the SUG model and its operations to a large collection of Maven 2 [5] and CRAN [6] super repositories. We then perform model operations on both the Maven and CRAN SUGs to demonstrate the usefulness of our models. By selecting different examples, we show different insights on adoption, diffusion and popularity within each universe. The Maven Central super repository is a major super repository that hosts many JVM project artefacts from the Java programming language ecosystem. Note that most projects in this repository are open-source Java, Scala or Clojure libraries (referred to as artefacts). R [25] is a free software environment for statistical computing and graphics. The Comprehensive R Archive Network (CRAN) belongs to an R ecosystem that hosts sources, binaries and packages related to the R environment. For the

Table 1: Construction of the SUG using attributes stored for Maven jar binaries and CRAN zip Windows Packages.

	Maven	**CRAN**
File type	POM.xml	DESCRIPTION
E_{use}	`<dependency>`	`Depends:`
E_{update}	`<version>`	`Version:`
$x.name$	`<groupId>.<artifactId>`	`Package:`
$x.release$	`<version>`	`Version:`
$x.time$	time-stamp of jar binary	indicated build time

Table 2: SUG Statistics for Maven Libraries and CRAN Windows Packages

	Maven	**CRAN**
Time Period	2005-11-03 to 2013-11-24	2003-08-29 to 2014-08-22
# of nodes	188,951	93,184
# of lineages	6,374	6,506
Reuse	5,146	1,517

experiment, we only targeted contributed packages under windows[4] (from R version 1.7 to 3.1). We speculate that other platforms (MacOX and Linux distributions) would yield similar results.

5 Empirical Results

5.1 SUG Construction and Analysis

Table 1 shows the different attributes collected from meta-files, that we used to construct the our SUGs. We constructed each SUG by extracting (i) use-relations, (ii) update-relations and (iii) software attributes such as the name, release and time properties of each software in the ecosystem. Specifically, we employed a typical extraction method of dependency information using meta-files [11,26–28]. For the Maven libraries, we use the Project Object Model file (i.e., `POM.xml`) that describes the project's configuration meta-data — including its compile-time dependencies[5], while each CRAN package stores its dependency information in a file called `'DESCRIPTION'`[6].

Table 2 presents a summary of data mined for the experiment. For our SUGs, our tools were able to mine and generate 188,951 Maven and 93,184 CRAN nodes from each super repository, spanning across 9-11 years. Note that independent software units (i.e.,without use-relation edges) were not included in SUGs. For the CRAN universe, we also include both available and archived packages.[7]. The SUGs were built from the dates shown in Table 2.

[4] `http://cran.r-project.org/bin/windows/contrib/`

[5] Refer to `http://maven.apache.org/pom.html` for the data structure

[6] `http://cran.r-project.org/doc/manuals/r-release/R-exts.html#The-DESCRIPTION-file`

[7] `http://cran.r-project.org/web/packages/` reports 5,833 at time of experiment

Table 3: Summary Statistics for L-SUG popularity

	Maven	**CRAN**
Min	1	1
1st Quartile	2	4
Median	6	8
Mean	38.8	64.62
3rd Quartile	20	22
Max	10160	26460

To understand the reuse within each universe, we measure how many lineages are being used internally. For every node u in SUG U:

$$reuse = \parallel \bigcup_{u \in U} UsedBy(u) \parallel \tag{13}$$

Table 2 details for each SUG. The results suggest that the CRAN universe reuse is comprised of a much smaller subset of reuse libraries (i.e., 1,518 lineages). On the contrary, in Maven there is an indication of more reuse within the universe (i.e., 5,146 lineages are used by 6,374 other lineages). We find this result not so surprising as most Maven artifacts comprise of either libraries or frameworks that may depend on multiple libraries.

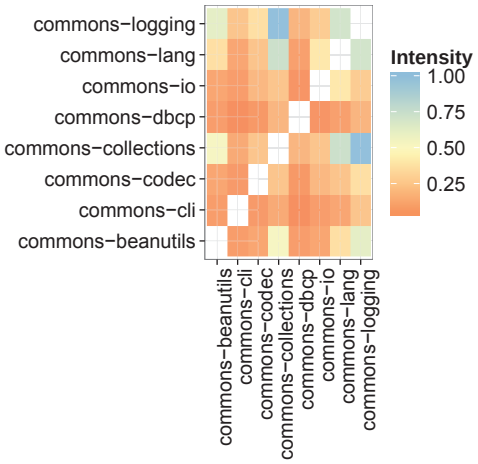
To determine popularity of a lineage, we apply the popularity function on a L-SUG. Hence, from the Maven and CRAN SUGs, we derive their respective L-SUGs with a lineage by lineage use relation. The statistical summary of this L-SUG popularity distribution for both Maven and CRAN is presented in Table 3. Surprisingly, we observe that except for the median and the 3rd quartile, both distributions of the Maven and CRAN universes seem similar. We had expected that since CRAN exhibited lower reuse (shown in Table 2), it would be more likely to generate higher popularity counts. For Maven, the testing library JUNIT was the most popular, while dependency on the R package release was found to be most popular in CRAN packages.

5.2 Coexistence Pairs using Heat Maps

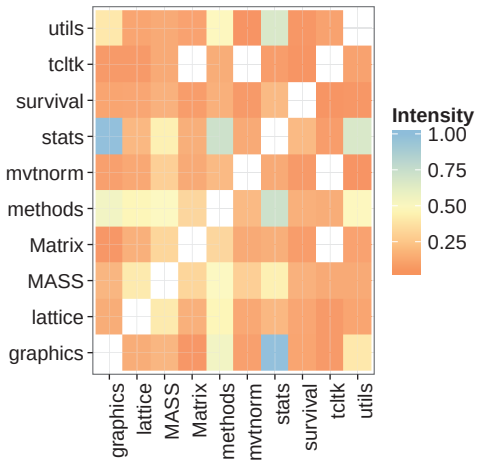
We utilize a heat map matrix to visualize coexistence pairs. We define the intensity as a normalized frequency count of popular pairs. For a given set of nodes I, for pairs $x, y \in I$ intensity is:

$$intensity(x, y, I) = \frac{popularity(x, y)}{\max\limits_{\substack{i,j \in I \\ i \neq j}} (popularity(i, j))} \tag{14}$$

where $x, y \in I$ and max returns the most frequent counts of pairing. The $Intensity$ function allows for normalized intensity shading of tiles from a scale of 0 to 1 (darker indicates higher popularity) by returning the most frequent pair count.



(a) Lineage pairings of eight Apache commons artifacts



(b) Top 10 popular CRAN package dependents (excluding R). Notice that pairs TCLTK, MATRIX and TCLTK, MVTNORM lack coexistence.

Fig. 6: Example of Coexistence Lineage Pairing

Using the L-SUG popularity, we utilize the heat map and the intensity function to plot popular pair frequency counts. Figure 6 illustrates examples of these lineage pairings. Figure 6(a) depicts the pairing of eight selected Maven Apache Commons [4] artifacts built for java. From the matrix, it is observed that the most popular pairing is between COMMONS-LOGGING, a log helper and COMMONS-COLLECTIONS, a library used for handling data structures. In Figure 6(b), we observe two cases of non coexistence. Figure 6(b) shows the top 10 most popular dependent packages[8] in CRAN. Notice that pairs (TCLTK, MATRIX) and (TCLTK, MVTNORM) lack coexistence. We speculate but cannot confirm that common functionality of handling complex matrices manipulation and GUI could be a reason. The pairing of (GRAPHICS,STATS) as the most frequent package combination is typical. This is because it is known as the 'free software environment for statistical computing and graphics' [25]. In summary, the results provide at a glance hints of popular pairings at the lineage level.

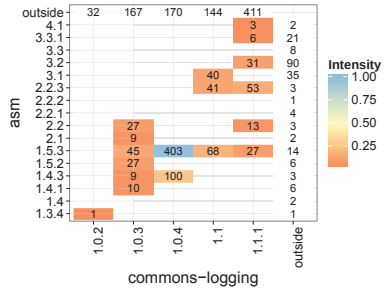
From the lineage pairing (L-SUG), popularity on the SUG is used to determine release pairs between two lineages. A heat map matrix with the intensity is used for this visualization. Figure 7(a), 7(b), 7(c) and 7(d) depicts the release pair plots between Maven's ASM [29], COMMONS-IO [30], COMMONS-LOGGING [31] and COMMONS-LANG [32]. Figures 7(e) and 7(f) show release pair plots between CRAN's MASS [33], MVTNORM [34] and LATTICE. Different to the L-SUG lineage pairs, the popularity is annotated at each pairing point. Additionally, the release pair plots include the popularity of a specific version x and any other software unit 'outside' $Lineage(y)$. It is plotted at the end of the x and y axis of the release pair plot. Formally, for each node n in U:

$$outside(x, y) = \sum_{\substack{n \in U \\ Lineage(n) \neq Lineage(x) \\ Lineage(n) \neq Lineage(y)}} popularity(x, n) \tag{15}$$

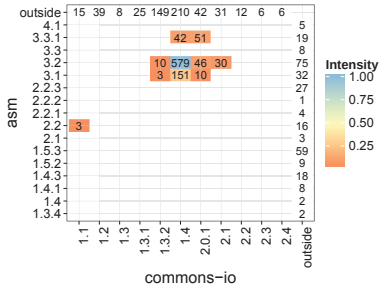
The *outside* pairs gauges relative popularity of alternative combinations. For instance, in Figure 7(b), the popularity of pairing Maven $\text{COMMONS-IO}_{1.4}$ and $\text{ASM}_{3.2}$ (popularity of 579) is greater than both 'outside' $\text{COMMONS-IO}_{1.4}$ (popularity of 210) and $\text{ASM}_{3.2}$ (popularity of 75). This indicated this pairing as very popular. On the other hand in Figure 7(e), CRAN's $\text{MASS}_{7.3-7}$ and $\text{MVTNORM}_{0.9.96}$ (popularity of 15) lower than the 'outside' $\text{MASS}_{7.3-7}$ (popularity of 411) indicate that the (MASS, MVTNORM) pair combination is not the most popular possible pairing combination.

Co-evolution patterns assist with interpretation of the evolution between the two lineages. As depicted in Figure 8, co-evolving patterns such as the *diagonal* co-evolving pattern can be identified from the heat map matrix. As shown in Figure 8, pairings with older releases can be easily distinguished. Figure 7(a) highlights $\text{COMMONS-LOGGING}_{1.0.3}$ and $\text{COMMONS-LOGGING}_{1.1.1}$ as popular pairings for ASM releases, forming an almost vertical pattern. While in Figure 7(b), there seems to be no apparent co-evolving pattern, with the

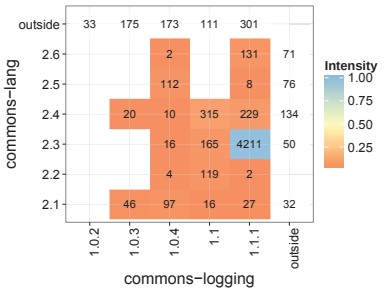
[8] excluding the R dependency



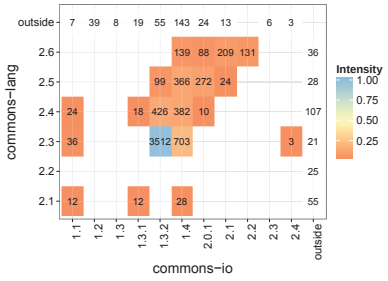
(a) ASM vs. COMMONS-LOGGING.



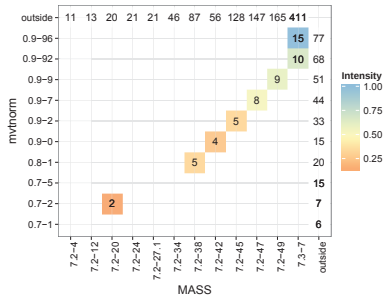
(b) ASM vs. COMMONS-IO.



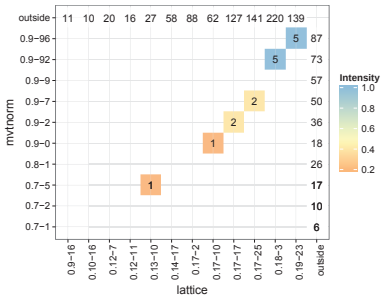
(c) COMMONS-LANG vs. COMMONS-LOGGING.



(d) COMMONS-LANG vs. COMMONS-IO.



(e) MASS vs. MVTNORM



(f) LATTICE vs. MVTNORM

Fig. 7: Example of Coexistence Mapping for both Maven Artifacts (a,b,c,d) and CRAN Packages (e,f). Note the different co-evolving patterns.

'crowd' settling on the $\text{ASM}_{3.2}$ and $\text{COMMONS-IO}_{1.4}$ pairing. Patterns from Figure 7(c) indicates that older releases of COMMONS-LANG have popular pairings even newer COMMONS-LOGGING releases. Figure 7(d) displays a more diagonal co-evolving pattern betwee COMMONS-LANG and COMMONS-IO.

Back to Figures 7(a), 7(b), 7(c) and 7(d) the intensity indicates that the most popular Maven 'crowd choices' are not necessarily the latest releases of their respective lineages. Maven maintainers seem conservative with older

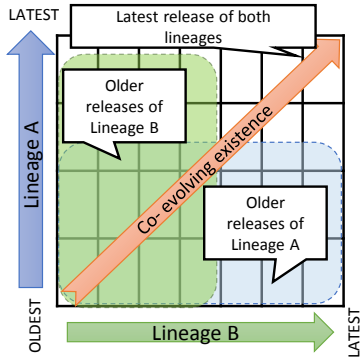


Fig. 8: Conceptual example of release pairs patterns, provide hints of popular pairings with older releases or if co-evolution (diagonal pattern) occurs.

Table 4: Top 10 filtered frequent counts of *coexistence rank* for lineage COMMONS-DBCP

lineage	*coexistence rank*
junit	37
commons-collections	28
log4j	28
commons-pool	27
commons-logging	25
hsqldb	20
commons-lang	15
derby	14
servlet-api	13
spring	13

versions. Quite to the contrary to the Maven pairings, CRAN packages depict a more distinguished diagonal pattern of co-evolution. This is illustrated in Figures 7(e) and 7(f). There are some inconsistencies such as non coexistence and overlapping pairs. Take for instance, the ($\text{MASS}_{7.3-7}$, $\text{MVTNORM}_{0.9-92}$) and ($\text{MASS}_{7.3-7}$, $\text{MVTNORM}_{0.9-96}$) pairings. This is due to the same version ($\text{MASS}_{7.3-7}$) released over two consecutive R releases. Also different to Maven, the most popular 'crowd choices' are the latest releases of their respective lineage. We conjecture that regular updates by maintainers to comply with the latest R release build check [9] may attribute to why CRAN packages are so well maintained. Also as outlined by Mens [35], since only a single version is allowed per release of R, maintainer must update to the compatible version or risk being removed in the next release. Regular updates for R packages may account for why the most popular releases being the most frequent release pairs.

Finally from the Maven super repository, we provide an example of a L-SUG *coexistence rank*. We perform the *coexist* function on a L-SUG of Maven, thus allowing a lineage as a node input. The operation returns lin-

[9] the latest build is accessible at `http://cran.rstudio.com/`

Table 5: Top 10 frequent counts of *coexistence rank* for lineage COMMONS-DBCP excluding popular lineages.

lineage	*coexistence rank*
mysql-connector-java	11
postgresql	10
c3p0	8
spring-test	7
cglib	6
aspectjweaver	4
bsh	4
geronimo-spec-jms	4
proxool	4
spring-web	4

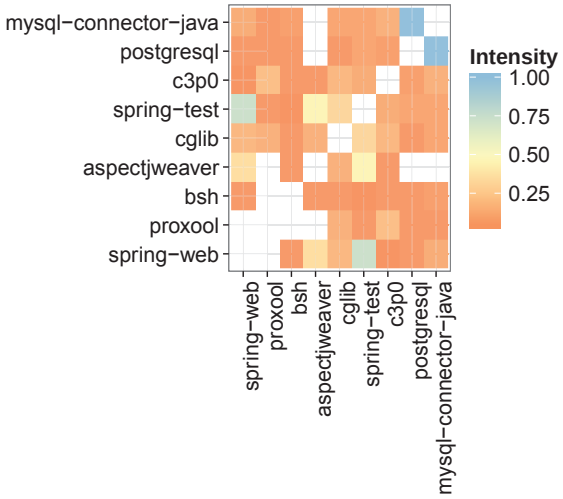


Fig. 9: Top COMMONS-DBCP filtered coexistence ranked lineage pairs

eages that at some point in time had a coexistence pairing. The Maven lineage artefact COMMONS-DBCP, a relational database connector [36] was used as an illustrative example. Using the Maven L-SUG, the *coexistence rank* returned 513 results. Table 4 pertains to the top 10 results. Note that the list comprises of many non database, general-purpose libraries such as JUNIT and LOG4J. For a more useful result, the use of L-SUG popularity thresholds (shown in Table 3) can be used to remove the more general purpose lineages, creating a more domain-specific result. We removed lineages that were above the 3rd quartile popularity threshold. The filtered results are provided in Table 5.

Compared to Table 4, the results in Table 5 provide more database specific entries such as MYSQL-CONNECTOR-JAVA[10], POSTGRELSQL[11], C3P0[12] and PROXOOL[13]. The filtered results are not entirely database specific, for instance the SPRING-WEB[14] library. As an extension, we can further explore lineage pairing of the domain-specific results in Table 5. Figure 9 depicts this result. Popular pairings of both POSTGRESL and MYSQL-CONNECTOR-JAVA suggests that in regards to COMMONS-DBCP, the postgresql and mysql databases are popular among the Maven 'crowd'. Note that in Figure 9, specialized library GERONIMO-SPEC-JMS[15] is not featured, indicating non coexistence with paring the results of Table 5. The non coexistence of MYSQL-CONNECTOR-JAVA and ASPECTJWEAVER[16] suggest that maintainers that depend on the ASPECTJWEAVER library should pay special attention to 'the wisdom of the crowd', as for one reason or another the crowd avoids its coexistence with the MYSQL-CONNECTOR-JAVA library. Note that the additional operations of filtering and heat map lineage pairing are examples of how users have the choice to use any combination of SUG operations.

5.3 Diffusion Plots

A key element lacking in the coexistence pairing visualizations is the SUG *temporal* properties. Diffusion Plots (DPs) are used to this end, so that adoption and diffusion of a lineage is seen at any point in time. For any lineage, we describe both $popularity_t$ and $variety_t$. For popularity, we plot the number of software units using a particular release of the lineage. Conversely in the variety plot, we track the number of lineages that use a specific release. Popularity is characterized as a curve that depicts important visual features, such as the steepness, when the curve halts and when the curve is *superseded* by a successive release curve. Figures 10 provides examples of DPs for both Maven and CRAN.

Figures 10(a) and 10(b) depict DPs for two lineages within the Maven Repository. COMMONS-LANG is a helper utility library, notably assisting with java string manipulation methods, basic numerical methods, object reflection, concurrency, creation and serialization and system. COMMONS-LOGGING is a java logger helper library. As seen in Figure 10(a), $\text{COMMONS-LANG}_{2.4}$ (dark green) it is overall the most popular although older release. Is is deemed as 'stable' by the 'crowd'. The steepness of the curve can be interpreted as the adoption trend. For instance in the $variety_t$ plot of Figure 10(b), we

[10] MYSQL-CONNECTOR-JAVA is a MySQL database java connector

[11] POSTGRELSQL is a PostgreSQL Driver JDBC4

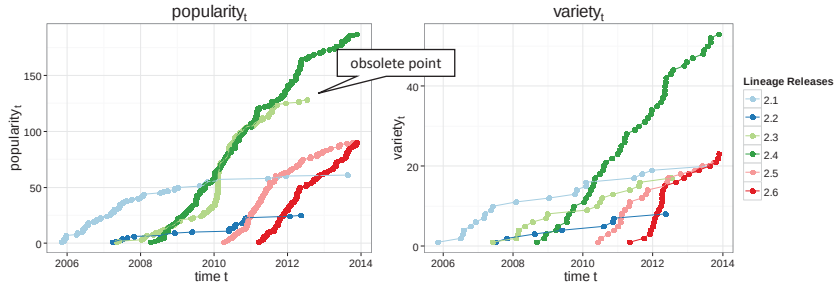
[12] C3P0 is a JDBC Connection pooling / Statement caching library

[13] PROXOOL is a Java connection pool.

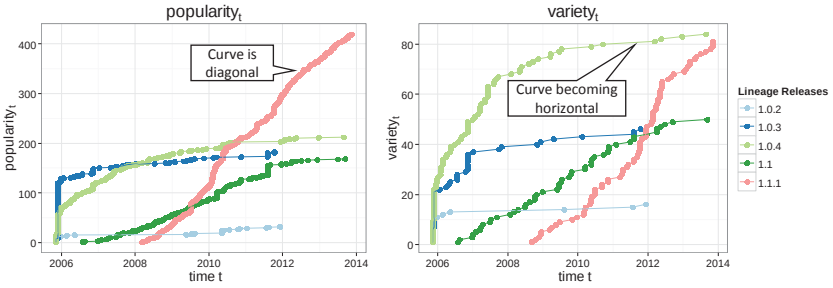
[14] SPRING-WEB is the web component of the spring framework

[15] GERONIMO-SPEC-JMS is Java Message Service (JMS) spec library for the Apache Genonimo application server

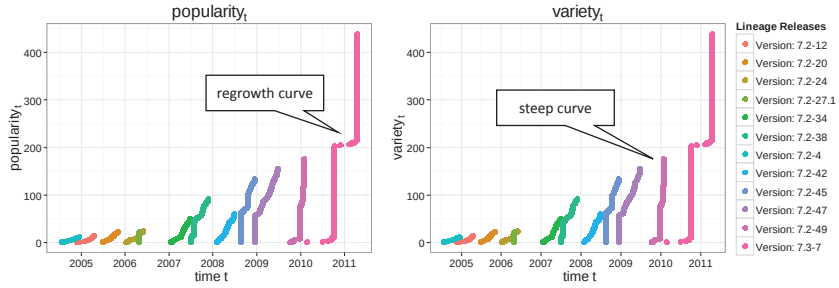
[16] ASPECTJWEAVER The AspectJ weaver introduces aspect oriented advices to java classes



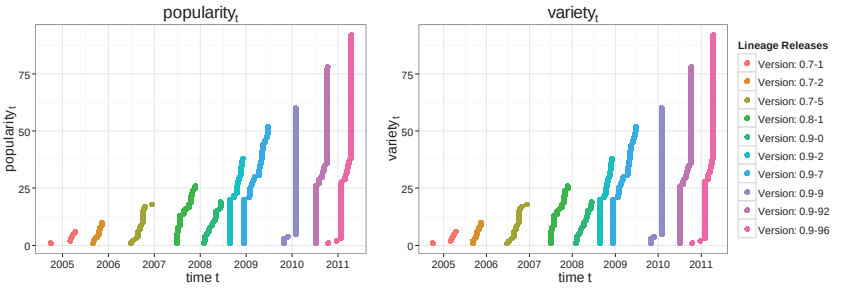
(a) Commons-lang



(b) Commons-loggings



(c) MASS



(d) mvtnorm

Fig. 10: Dependents Diffusion Plots for selected Maven and CRAN lineages. The left hand depicts the $popularity_t$ while the right shows the $variety_t$ for their respective releases.

observe that COMMONS-LOGGING$_{1.0.4}$ has the most variety at any point in time. However, closely looking at its curve (light green), popularity has probably peaked with the curve almost horizontal. COMMONS-LOGGING$_{1.1.1}$ (pink), however, adopts a more diagonal curve, hinting future adoptions could follow this trend. Note that the predecessor COMMONS-LANG$_{2.4}$ (dark green) in Figure 10(a) is still adopted beyond the COMMONS-LANG$_{2.3}$ obsolete point, making it more successful. Significant difference between $popularity_t$ and the corresponding $variety_t$ indicate lineages with abnormally high releases depend on this specific lineage release. For instance, in Figure 10(a), on the `y-axis` depicts $popularity_t > 150$ while $variety_t > 40$. Such variances could be misleading, so should be considered.

The DPs of the CRAN universe exhibit a much different adoption and diffusion behavior. Figures 10(c) and 10(d) show the DPs of both MASS and MVTNORM lineages. The Mass package supports functions and datasets for Venables and Ripley's MASS while the MVTNORM package computes multivariate normal and t probabilities, quantiles, random deviates and densities. Depicted in Figure 10(c) and 10(d), with regular superseding of the previous release. Previous release curves never overlap but grow incrementally over time. Note that the most popular library versions deemed by the 'crowd' are the latest releases. Differences in corresponding $popularity_t$ and $variety_t$ values are minimal. This is consistent with the fact that only a single lineage release is permitted per R release. The steepness of the rate of adoption is almost vertical, meaning migration to newer releases were almost instantaneous, while making previous releases obsolete. In Figure 10(c), the $Mass_{7.3-7}$ curve follows an abnormal *regrowth type* shape possibly caused by migration to a newer R release. In summary, DPs offer a visualization of adoption and diffusion patterns for a lineage. This has shown to be particularly useful in the case of the Maven universe, where adoption is slow and older releases still popular. Particularly, results in Figure 10(a) conincide with Figure 7(c), where the older releases of COMMONS-LANG are still preferred newer versions. Superseding points may provide hints on when safe to update. Additionally as shown, care must be taken in the interpretation when there is a variance in $popularity_t$ and $variety_t$.

6 Discussion

6.1 SUG Model Considerations

The SUG model provides for a standardized comparison of use-relations across diverse structured universes of software repositories within the ecosystem. The model allows for uncovering adoption, diffusion and popularity within a super repository universe, that is useful for assessing dependency management issues for developers. In this study, we only looked at two different ecosystems, however, we would like to investigate more ecosystems such as the 'C programming language' libraries universe. As the rise of cross-language (or platform) sys-

tems emerge, potential future research avenues could be the adaptation from a different universe and how system applications deal with cross-universe interaction. The use of an abstract model has its drawbacks, such as disregarding more complex concepts such as package 'containment' and 'transitive' relationships. At this stage we only consider the basic elements of software usage to measure popularity. As we study more systems, we will consider modeling additional concepts common to all universes as extensions to the model. A concern in the SUG construction was the assignments of the R_{use} and R_{up} edges in relation to the name attribute of the node. Threats specific to the Maven and CRAN repositories, such as changed domains or lineage from more multiple nodes exist. Nevertheless, conceptually the SUG model was successfully applied to each universe. For future improvements of accuracy the model can expand beyond the name attribute for lineage classifications. We plan to incorporate more sophisticated techniques and tools used in 'code clone' such as code clone detection [37,38] and 'origin' analysis [39,40] to determine common lineage. Recently, Ishio et al. presented a promising technique to find software provenance using its Software Bertillonage [41].

There exists many definitions of software variability and dependency relationships. In Software Product Line Engineering (SPL), terms such as 'product' variability has been used extensively [42–44]. In the code clones field, Kim et al. [45] coined clone 'genealogies' to track variability between software of similar origins. To coincide with the abstract nature of the SUG and avoid preconceptions, we decided on the term 'lineage' to describe variability between software releases. In addition, systems and libraries are not explicitly distinguished. Graph cyclic based approaches such as ranking (such as page ranking), reference counting and component ranking is common for measuring popularity on graph based models [11],[46]. Our SUG model does not employ any of these approaches as it is designed to rely on the dependency chains but to measure immediate dependency characteristics directly.

6.2 Model Operations

As stated in our background, related studies have all reported maintainer's concerns with API breakages and incompatibilities of existing dependencies over time. The results of this study revealed that the coexistence mapping provides interesting visual patterns. Assuming that usage implies stability, we can identify safe combinations of lineages deemed by the 'crowd'. The coexistence operations on the SUG demonstrate more 'intuitive' aspects of the model, although domain specific filtering may be required. Another complex but useful operation that was not presented in this paper is the tracing of systems that have abandoned a dependency. This could be future work. The Diffusion Plots (DPs) provide for a more temporal analysis of popularity and diffusion. Consistent with the SUG temporal properties, the restriction of adding nodes results in incremental adoption curves. The accumulating growth provides for a comparison of previous releases. Overall the DPs describe different

adoption behaviors within the universe. Our qualitative visualizations indicate that maintainers of CRAN packages are more inclined to update to the latest version of their dependencies. The conservative nature of Maven artifact maintainers on the other hand, further justifies the potential usefulness of the SUG. Its model querying visualizations should assist maintainers to gain intuitive insights and understand the opportunities for updating components. We envision that an integration of both the coexistence and DP temporal properties onto a single visualization would be beneficial. This is seen as future work.

In practice, updates based solely on popularity is often not practical as there are so many factors to consider. Therefore for a complete toolkit, usage popularity should be complementary to the many existing code and API compatibility checking tools. Ultimately, the maintainer's personal preference may override all options. Our aim is to increase the maintainers knowledge by making aware potential insights that assist the maintainer to make the right choice.

6.3 Threats to Validity

The main internal threat to validity is the real-world assessment of the usefulness of our model. We believe that we are currently at conceptual stage of the research and this is seen as future work. We have been working closely with system integration industrial partners to develop and test our visualizations. We have received positive feedback regarding the modeling and particularly concerning coexistence. Another threat is that the results that we show may not be generalized for all types of systems. At this stage we have studied other systems but we are confident that current trends hold. We envision that only further querying and usage we refine our results, thus future improvements will be endeavored. We understand that in reality that the the repository data can be modified such as when changing domain locations, thus threatening the temporal property. Our investigations have proven, however, that most SUGs hold the temporal property.

An external threat is that our datasets only includes information about dependencies that are explicitly stated in project configuration files, such as the Maven `POM` and R `Description` files. It does not take into account reuse such as copy-and-paste and clone-and-own. Although gauging dependencies by the configuration file only provides for a sample of the actual reuse, we believe this is sufficient to give an impression of trends within each universe. We understand that our data and analysis are dependent on the tools and analysis techniques. Threats include parsing techniques. However, we believe that our samples are large enough to be representative of the real world.

7 Related Work

Next to work discussed in Section 2 and Section 6.1, work from the following research domains is complementary:

7.1 Library Popularity Measures

Raemaekers et al. [10, 47, 26] performed several studies on the Maven repositories about the relation between usage popularity and system properties such as size, stability and encapsulation. Popularity has also been leveraged in IDEs. For instance, Eisenberg et al. improve navigation through a library's structure using the popularity of its elements to scale their depiction [20]. In this work, we introduce the SUG model as a means to utilize the popularity and diffusion measures. We define more complex popularity measures and visualizations through an abstract model.

7.2 Code Search and Recommendations

Code search is prominent among research on software reuse with many benefits for system maintainers [48]. Examples of available code search engines are google code [49] and ohloh code search [50]. Tools such as Ichi-tracker [51], Spars [46], MUDAblue [52] and ParserWeb [53] just a few of the many available search tools that crawl software repositories mining different software attributes and patterns with different intentions. For instance, SpotWeb searches for different library usage patterns while MUDAblue automatically categorizes related software systems.

We also crawl the super repositories, using mined data to construct our abstract SUG Models. Differently, our work involves purely popularity metrics to locate through model operations and visualization different coexistence and diffusion behavior.

7.3 Software Systems as Components Within an Ecosystem

Recently, there has been an increase in research that perceives software systems as being components that interact and form dynamic relationships within an ecosystem. Work such as Bosch [42] have studied the transition from Product Lines to an Software Ecosystem level of abstraction. German et al. [27] studied the GNU R project as an ecosystem over time. Since the projects inception, the results of the study indicate that user-contributed systems have been growing faster than core-systems and identified differences of how they attracted active communities. Bogart et al. [12] studied how different ecosystems deal with API changes of their evolving libraries. The results show differences in each ecosystem policy and its supporting infrastructure; and there is value in making community values and accepted tradeoffs explicit and transparent in order to resolve conflicts and negotiate change-related costs. Furthermore, Mens et al. [54] perform ecological studies of open source software ecosystems with similar results. Haenni et al. [55] performed a survey to identify the information that developers lack to make decisions about the selection, adoption and co-evolution of upstream and downstream projects in a software ecosystem. In

this context, we consider that the SUG provides insights into the ecosystem of a particular software universe. We suggest that the different visualizations and patterns can complement the work and provide insights into the ecological structure.

8 Future Work and Conclusion

With the emergence of Maven, CRAN, and GitHub super repositories, opportunities arise to uncover insights valuable to the management of dependencies through intelligent super repository mining. In this paper, we present the SUG model as a means to represent, query and visualize different super repositories in a standardized and systematic manner. Immediate future work focuses on evaluating the insigthfullness of these queries and visualizations with actual system maintainers. Our work is towards empowering maintainers to make more informed decisions about whether or not to update the library dependencies of a system. Combining its "wisdom-of-the-crowd" insights with complementary work on compatibility checking of API changes, should give rise to a comprehensive recommendation system for dependency management.

References

1. C. Ebert, "Open source software in industry," in *IEEE Software*, 2008, pp. 52–53.
2. L. Hainemann, F. Deissenboeck, M. Gleirscher, B. Hummel, and M. Irlbeck, "On the extent and nature of software reuse in open source java projects," in *Proceedings of the 12th International Conference on Top Productivity Through Software Reuse*, 2011, pp. 207–222.
3. Spring framework homepage, accessed 2014-09-01. [Online]. Available: http://spring.io/
4. Apache commons library homepage, accessed 2014-09-01. [Online]. Available: http://commons.apache.org/
5. Maven central repository, accessed 2014-09-01. [Online]. Available: http://mvnrepository.com/
6. Comprehensive r archive network(cran), accessed 2014-09-01. [Online]. Available: http://cran.rstudio.com/
7. Sourceforge repository, accessed 2014-09-01. [Online]. Available: http://sourceforge.net/
8. Github repository, accessed 2014-09-01. [Online]. Available: https://github.com/
9. R. E. Grinter, "Understanding dependencies: A study of the coordination challenges in software development," *Ph.D. Thesis. University of California. Department of Information and Computer Science.*, 1996.
10. S. Raemaekers, A. van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *Proc. of Intl. Comf. Soft. Main. (ICSM)*, Sept 2012, pp. 378–387.
11. C. Teyton, J.-R. Falleri, and X. Blanc, "Mining library migration graphs," in *Proc. of. Work. Conf. on Rev. Eng. WCRE2012*, Oct 2012, pp. 289–298.
12. C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an api: Cost negotiation and community values in three software ecosystems," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, 2016, pp. 109–120.
13. R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to api deprecation?: The case of a smalltalk ecosystem," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 56:1–56:11.

14. A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, "How do developers react to api evolution? the pharo ecosystem case," in *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ser. ICSME '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 251–260.
15. A. A. Sawant, R. Robbes, and A. Bacchelli, "On the reaction to deprecation of 25,357 clients of 4+1 popular java apis," in *Proceedings of the 32th IEEE International Conference on Software Maintenance and Evolution.*, 2016.
16. G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the apache community upgrades dependencies: An evolutionary study," *Empirical Softw. Eng.*, vol. 20, no. 5, pp. 1275–1317, Oct. 2015.
17. R. G. Kula, C. D. Roover, D. M. German, T. Ishio, and K. Inoue, "Visualizing the evolution of systems and their library dependencies," *Proc. of IEEE Work. Conf. on Soft. Viz. (VISSOFT)*, 2014.
18. S. Chuan-Fong and V. Alladi, "Beyond adoption: Development and application of a use-diffusion model," *Journal of Marketing*, 2004.
19. R. Holmes and R. J. Walker, "Informing Eclipse API production and consumption," in *OOPSLA2007*, 2007, pp. 70–74.
20. D. S. Eisenberg, J. Stylos, A. Faulring, and B. A. Myers, "Using association metrics to help users navigate API documentation," in *VL/HCC2010*, 2010, pp. 23–30.
21. 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.
22. Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining trends of library usage," in *ERCIM Workshops*, 2009, pp. 57–62.
23. Y. M. Mileva, V. Dallmeier, and A. Zeller, "Mining API popularity," in *TAIC PART*, 2010, pp. 173–180.
24. R. Bloemen, C. Amrit, S. Kuhlmann, and G. Ordóñez Matamoros, "Innovation diffusion in open source software: Preliminary analysis of dependency changes in the gentoo portage package database," in *Proc. of Work. Conf. on Mining Soft. Repo. (MSR2014)*, 2014, pp. 316–319.
25. R statistical computing and graphics project homepage, accessed 2014-09-01. [Online]. Available: http://www.r-project.org/
26. S. Raemaekers, G. Nane, A. van Deursen, and J. Visser, "Testing principles, current practices, and effects of change localization," in *Mining Soft. Repo. (MSR)*, May 2013, pp. 257–266.
27. D. M. German, B. Adams, and A. E. Hassan, "The evolution of the r software ecosystem," *Proc. of European Conf. on Soft. Main. and Reeng. (CSMR2013)*, pp. 243–252, 2013.
28. Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining trends of library usage," in *Proc. Intl and ERCIM Principles of Soft. Evol. (IWPSE) and Soft. Evol. (Evol) Workshops*, ser. IWPSE-Evol '09. New York, NY, USA: ACM, 2009, pp. 57–62.
29. Apache commons asm homepage, accessed 2014-09-01. [Online]. Available: http://asm.ow2.org/
30. Apache commons io homepage, accessed 2014-09-01. [Online]. Available: http://commons.apache.org/proper/commons-io/
31. Apache commons logging homepage, accessed 2014-09-01. [Online]. Available: http://commons.apache.org/proper/commons-logging/
32. Apache commons lang homepage, accessed 2014-09-01. [Online]. Available: http://commons.apache.org/proper/commons-lang/
33. Cran mass package homepage, accessed 2014-09-01. [Online]. Available: http://cran.r-project.org/web/packages/MASS/index.html
34. Cran mvtnorm package homepage, accessed 2014-09-01. [Online]. Available: http://cran.r-project.org/web/packages/mvtnorm/index.html
35. M. Claes, T. Mens, and P. Grosjean, "On the maintainability of CRAN packages," in *Proc. of CSMR-WCRE 2014*, 2014, pp. 308–312.
36. Apache commons dbcp homepage, accessed 2014-09-01. [Online]. Available: http://commons.apache.org/proper/commons-dbcp/
37. C. K. Roy and J. R. Cordy, "A survey on software clone detection research," in *Technical Report No. 2007-541,Queens University, Canada*, 2007.

38. T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.
39. M. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 166–181, 2005.
40. J. Davies, D. M. German, M. W. Godfrey, and A. Hindle, "Software bertillonage: Finding the provenance of an entity," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 183–192.
41. T. Ishio, R. G. Kula, T. Kanda, D. M. German, and K. Inoue, "Software ingredients: Detection of third-party component reuse in java software release," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16, 2016, pp. 339–350.
42. J. Bosch, "From software product lines to software ecosystems," in *Proc.of the Int Soft. Prod. Line (SPLC '09)*, 2009, pp. 111–119.
43. C. Seidl and U. Assmann, "Towards modeling and analyzing variability in evolving software ecosystems," in *Proc. of the Int. Workshop on Variability Modelling of Software-intensive Systems (VaMoS '13)*, 2013.
44. M. Nonaka, K. Sakuraba, and K. Funakoshi, "A preliminary analysis on corrective maintenance for an embedded software product family," *IPSJ SIG Technical Report*, vol. 2009-SE-166, no. 13, pp. 1–8, 2009.
45. M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th International Symposium on Foundations of Software Engineering*, 2005, pp. 187–196.
46. K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking significance of software components based on use relations," *Software Engineering, IEEE Trans.*, vol. 31, pp. 213–225, March 2005.
47. S. Raemaekers, A. v. Deursen, and J. Visser, "The maven repository dataset of metrics, changes, and dependencies," in *Proc. Conf. on Mining Soft. Repo.*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 221–224.
48. S. Bajracharya, A. Kuhn, and Y. Ye, "Proc. of work. on search-driven dev.: Users, infrastructure, tools, and evaluation (suite 2011)," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011.
49. Google code, accessed 2014-09-01. [Online]. Available: https://code.google.com/
50. Ohloh code search, accessed 2014-09-01. [Online]. Available: https://code.ohloh.net/
51. K. Inoue, Y. Sasaki, P. Xia, and Y. Manabe, "Where does this code come from and where does it go? - integrated code history tracker for open source systems -," in *Proc. of Inl Conf. on Soft. Eng.*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 331–341.
52. S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "MUDABlue: an automatic categorization system for open source repositories," *Journal of Systems and Software*, vol. 79, no. 7, pp. 939–953, 2006.
53. S. Thummalapenta and T. Xie, "Parseweb: A programmer assistant for reusing open source code on the web," in *Proceedings of the IEEE/ACM Intl. Conf on ASE*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 204–213.
54. T. Mens, M. Claes, and P. Grosjean, "Ecos: Ecological studies of open source software ecosystems," in *Soft. Main. Reeng. and Rev. Eng. (CSMR-WCRE)*, Feb 2014, pp. 403–406.
55. N. Haenni, M. Lungu, N. Schwarz, and O. Nierstrasz, "Categorizing developer information needs in software ecosystems," in *Proc. of Int. Work. on Soft. Eco. Arch. (WEA13)*, 2013, pp. 1–5.