Technical Report of Osaka University, Department of Computer Science, Graduate School of Information Science and Technology

# Modeling Library Dependencies and Updates in Large Super Repository Universes

Raula Gaikovina Kula[a], Coen De Roover[b], Daniel M. German[c], Takashi Ishio[b], Katsuro Inoue[a]

Osaka University, Japan[a]
Vrije Universiteit Brussel, Brussels, Belgium[b]
University of Victoria, Canada[c]

Technical Report: 11092015-SEL

Raula G. Kula, Takashi Ishio and Katsuro Inoue are with the Software Engineering Laboratory, Osaka University, Japan. E-mail: {raula-k, ishio, inoue} @ist.osaka-u.ac.jp. Coen De Roover is with the Software Languages Lab, Vrije Universiteit Brussel, Brussels, Belgium. E-mail: cderoove@vub.ac.be. Daniel M. German is with the University of Victoria, Canada. E-mail: dmg@turingmachine.org

# Modeling Library Dependencies and Updates in Large Super Repository Universes

Raula Gaikovina Kula[a,*], Coen De Roover[b], Daniel M. German[c], Takashi Ishio[a], Katsuro Inoue[a]

[a] *Osaka University, Japan*
[b] *Vrije Universiteit Brussel, Brussels, Belgium*
[c] *University of Victoria, Canada*

## Abstract

Popular (re)use of third-party open-source software (OSS) provides evidence of how large library hosting super repositories (like maven central) influence the software development world today. Updating libraries is crucial, with recent studies highlighting risks and vulnerabilities with aging OSS libraries. Decisions to adopt a newer library can range from trivial (security threat) to complex (assessment of work required to accommodate the changes). By leveraging the 'wisdom of the super repository crowd', we propose a simple and efficient approach to recommending 'consented' library updates. Our Software Universe Graph (SUG) models library dependency and update information mined from super repositories. To evaluate, we first constructed a SUG from with 1.6 million nodes of about over 37,600 unique maven artifacts. Results show that although most projects have a high reuse tendency, most projects have less popularity across the repository. We found on median average, each project has 2 dependencies and 5 other projects that are dependent on it. Secondly, as a case study, we apply our SUG metrics to two real-world examples. Our adoption-diffusion profiling depict advanced popularity perspectives. We envision our SUG model can be extended to allow for more recommendations such as replacement or new libraries.

*Keywords:* Library Reuse, Mining Software Repositories, Empirical Studies

*Raula G. Kula is with the Software Engineering Laboratory, Osaka University, Japan. E-mail: raula-k@ist.osaka-u.ac.jp.

## 1. Introduction

The (re)use of third-party software is now commonplace in today's software development, both open source software (OSS) and commercial settings alike [1], [2]. Software libraries come with the promise of being able to reuse quality implementations, 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. Widespread use of OSS libraries has lead to massive stores of project repositories such as The Central Repository (Maven) [5], Sourceforge [6] and Github [7]. For instance, as of October 5th 2015, Maven (`https://search.maven.org/#stats`) hosted over 120,000 unique projects.

Software is constantly evolving. With new versions continuously released, the maintenance of system's dependencies is not practiced enough. A study by Grinter identified aging libraries a threat to software livelihood [8]. In 2014, Sonatype reported that on average 24% of buggy code in applications were linked to severe flaws in their outdated libraries. That same year, the threat of high profile vulnerabilities Shellshock[1], HeartBleed[2] and Poodle[3] highlighted the need to update dependencies in applications (also referred to as systems in this paper). Security vulnerabilities updates are a trivial decision as its threat to software quality outweighs the costs. Security experts recommend to update, regardless of the size of the changes to be made.

More complex decisions are encountered when assessing the different risks and effort required to accommodate the changes. Many studies [9, 10, 11], have reported that unless the underlying need is apparent, most maintainers are unmotivated or hesitant to update. Our previous work [12] considered that developers exhibit a latency to migrate to the latest version released.

To this end, tools and techniques have been developed to address certain risks of migration. Take for instance, library incompatibility. Research tools such as `SemDiff` [13] and industry counterparts like `clirr` [14] are used to assist with library compatibility issues during migration. Moreover, other external technical, organizational or social factors also influence a maintainers decision to update. For instance, a maintainers personal preference or compliance to the organizational practices may influence the decision. These

[1]`https://shellshocker.net/`
[2]`http://heartbleed.com/`
[3]`https://poodlebleed.com/`

techniques though effective, only solve a specific risk.

With the advancements in online repository usage and data mining, we provide a much more efficient and simpler solution to library update recommendations. Building on our previous work on visualizing the evolution of a system and its library dependencies [15] and on popular dependency combinations [16], we introduce the Software Universe Graph (SUG) as a generic means to model and quantify "wisdom-of-the-crowd" insights for a software repository universe. We extend on the simple usage popularity metric with metrics to describe *adoption-diffusion*. Our popularity is a measure of usage at any point in time. The SUG is used to profile *adoption-diffusion* characteristics of library versions over the super repository. For the evaluation, we show a real-world construction of a SUG , then through adoption-diffusion profiling demonstrate practical library migration recommendations. The profiles show 1.) distinction between popularity among a small set of projects and popularity across the whole super repository, 2.) older versions can be still popular, 3.) and we can predict trends of attractive versions. The paper makes the following contributions:

- We introduce the graph-based SUG model to represent library dependency and update relationships within a large-scale super repository universe. We demonstrate practicality by construction from maven.
- We use the SUG to present the notion of adoption-diffusion profiles for a project. We use real-world system to demonstrate usage.
- We show through adoption-diffusion profiles, the SUG can provide practical library migration recommendations.

The paper layout is as follows. Section 2 details the motivation of the SUG. Section 3 explains in detail the formal aspects of the SUG model. Section 4 introduces the metrics applied to the SUG model. Section 5 discusses the evaluation with the results presented in section 6. Discussions and related works are later shown in Section 7 and 8 respectability. Finally, we close with conclusions in Section 9.

## 2. Mining the 'wisdom of the crowd' from Super Repositories

Our approach involves studying the different library dependency relationships that exist in the super repository over time. Concretely, we are concerned with the diffusion of newer libraries. According to the Diffusion of Innovations (DoI) theory [17], successful technologies have different types of users: innovators, early adopters, the early majority, the late majority, and laggards. Applied to the super repository dependency relationships, we would like to understand the diffusion in terms of popular migration toward the different versions of libraries. Our rational is that crowd 'consent' of a library is evident by its successful adoption and diffusion over its predecessors. Our adoption-diffusion concept is inspired by use-diffusion [18] metrics used in the field of economics and marketing.

The changes in the complex web of dependency relationships in the super repository characterizes the ripping effect of updating a single library dependency. The colloquial term 'dependency hell', to describe these complexity of managing these dependencies. Maven and Gradle[4] are examples of dependency management build tools employed for applications. In this paper, we formulate a model in which adoption-diffusion relationships can be captured, quantified and visualized using defined metrics of popularity and adoption-diffusion. Using a graph-based approach, we model dependency and update relations to handle all software systems in a super repository.

## 3. The Super Repository Universe

### *3.1. Modeling Super Software Repositories*

In this section, we show in Figure 1 how our model handles the realities of library dependencies and update across software repositories. We consider the virtual repository universe that encompasses both publicly accessible and private repositories. We define a *project release* as a published software unit with a version identifier. For instance, version `3.6.3` of `SymmetricDs` ($SymmetricDs_{3.6.3}$). A project release is either in source or in executable format. Examples of language-specific source code are `*.java, *.cpp, *.jss` accompanied by configuration build files. Executables are compiled binaries such as `jar, exe or dll` files ready for (re)use. A project release may be superseded by a newer project release, creating an update relationship. Project releases can use other project releases as libraries and vice-versa, forming a dependency relation. Project releases linked by update relationships are managed by a *project repository*. Project repositories may manage

[4]`http://gradle.org/`

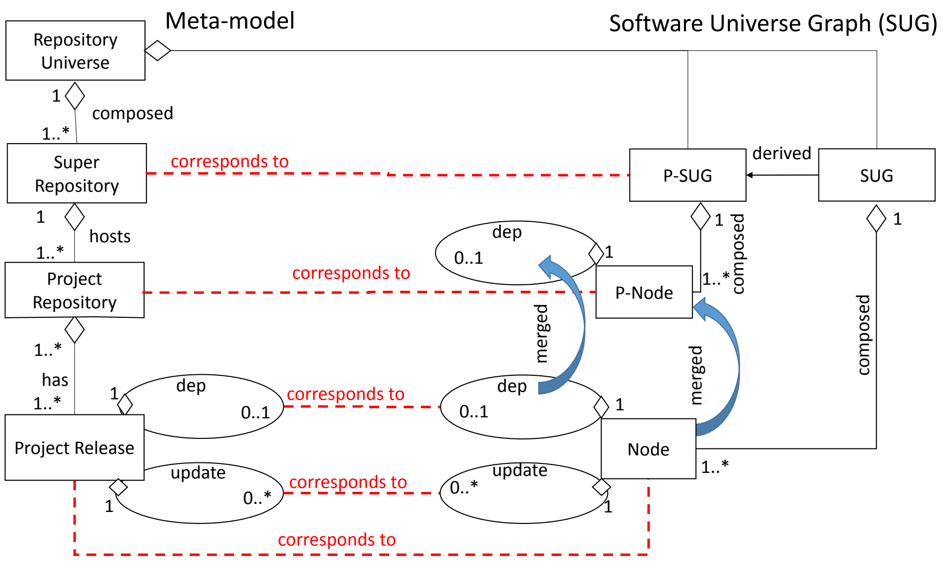
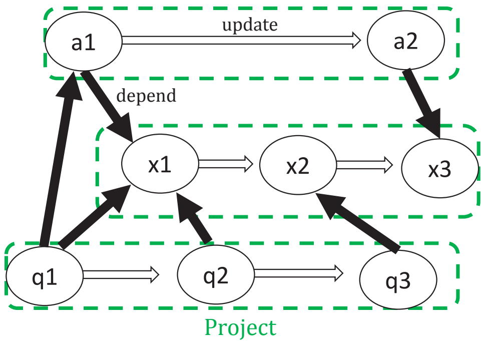


**Figure 1:** Meta-model of the real world and the proposed SUG model.

project release relations through project-specific conventions such as Semantic Versioning (SemVer)[5]. The super repository hosts multiple project repositories. Related work refers to these as 'super' repositories or repositories of repositories [19], [20]. We discern two types of super repositories: those that host libraries and those that host systems. Examples of library-hosting super repositories include MAVEN for JVM libraries, RUBYGEMS[6] for Ruby libraries, and NUGET[7] for .NET and NPM[8] for JavaScript libraries. Examples of system-hosting super repositories include GITHUB and SOURCEFORGE which primarily serve as hubs for collaborative development and end-user download respectively.

As depicted in Figure 1, the SUG is an abstract representation of the realities of super repositories. Related studies reveal web-like complex dependencies between project releases, making the distinction between systems and libraries dependent on perspective [21], [22]. Dependencies can even span

[5]`http://semver.org`
[6]`https://rubygems.org`
[7]`https://www.nuget.org`
[8]`https://www.npmjs.com`



$depend(a1) = \{x1\}$

$dependsOn(x1) = \{a1, q1, q2\}$

$Project(q1)$
$= Project(q2) = Project(q3)$
$= \{q1, q2, q3\}$

$popularity\ (x1)$
$= |rev\text{-}depend\,(x1)|$
$= |\{a1, q1, q2\}| = 3$

$variety(x)$
$= \|rev\text{-}depend\,(x)\| = ||a1, q1, q2||$
$= |\{Project(\text{a}), Project(\text{q1}), \text{Project(q2)}\}|$
$= 2$

**Figure 2:** Example of the Software Universe Graph

.

across super repositories. Therefore, the model should not be restrictive in system or library identification. The model should also not be restrictive in implementation issues such as programming language and control version systems. Specifically there are two types of software universe models, the normal SUG (introduced in Section 3.2) that corresponds to the project releases and the P-SUG that correponds to dependencies at the project repository level. The P-SUG is an aggregation of nodes and edges (merged dependency edges and dropped update edges) related to one particular project repository (later introduced in Section 3.3).

## *3.2. The Software Universe Graph (SUG)*

Figure 2 depicts the basic elements of the Software Universe Graph. Let $G(N, E)$ be a SUG. $N$ is a set of nodes, with each node representing a project release instance. For instance, `SymmetricDs` version `3.6.3` ($SymmetricDs_{3.6.3}$) is a project release instance represented as a single node. For any SUG, the edges $E$ are composed of $E_{dep}$ and $E_{up}$. $E_{dep}$ is a set of *dependency* edges and $E_{up}$ is a set of *update* edges.

**Definition 1.** *An edge $u \rightarrow v \in E_{dep}$ means that $u$ depends on $v$ (depend). Reverse-dependency (rev-depend) refers to the inverse.*

$$depend(u) \equiv \{v | u \rightarrow v\} \tag{1}$$

$$rev\text{-}depend(u) \equiv \{v | v \rightarrow u\} \tag{2}$$

Dependency-relations can be extracted from either the source code or from build configuration files. As depicted in Figure 2, node $a1$ (system) has a depend relation to node $x1$ (library). Note that node $x1$ has reverse dependencies (rev-depend) to nodes $a1$, $q1$ and $q2$. Parallel edges for node pairs are not allowed. In this paper, we focus on popular project releases that are connected by many depend-relation edges.

**Definition 2.** *For a given node u, popularity is the number of incoming depend-relation edges.*

$$popularity(u) \equiv |rev\text{-}depend(u)| \tag{3}$$

For instance in Figure 2, for node $x1$, $popularity(x1) = |rev\text{-}depend(x1)| = |\{a1, q1, q2\}| = 3$.

**Definition 3.** *An edge $a \Rightarrow b \in E_{up}$ represents an update-relation from node $a$ to $b$, meaning $b$ is the immediate successor release of $a$.*

Update-relations refer to when a succeeding release of a project release is made available. Figure 2 shows that node $q1$ is first updated to node $q2$. Later on, node $q2$ is updated to the latest node $q3$. Hence, $q1 \Rightarrow q2 \Rightarrow q3$.

Let any SUG node $u$ be denoted by three attributes: `<name, release, time>`. For a node $u$, we define:

- **u.name** Name is the string representing the identifier of a software project.

  For nodes $x$ and $y$, if $x \Rightarrow y$, then $x.name = y.name$ holds in the SUG.

- **u.release**. Release denotes a version reference for a software project. For nodes $u$ and $v$, if $u \Rightarrow v$ then $v$ is the immediate successor of $u$.

- **u.time**. Time refers to the time-stamp at which node $u$ was released. For nodes $x$ and $y$ of $x \Rightarrow y$, $x.time < y.time$.

The SUG node for a release [9] of JUNIT, for instance, is `<name = "junit", release= "4.11", time="2012-11-14">`.

[9]`http://mvnrepository.com/artifact/junit/junit/4.11`: accessed 2014-08-02

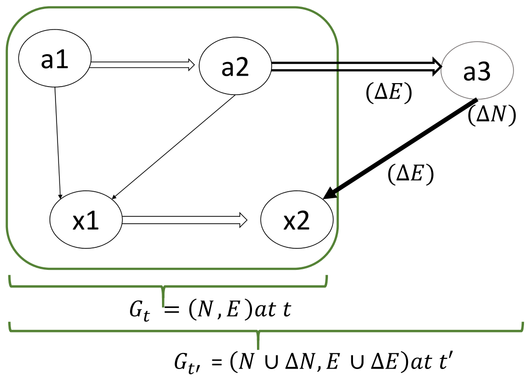


**Figure 3:** Temporal property of the SUG

**Property 1.** *A SUG monotonically increases its nodes and edges.*

Let SUG $G_t = (N, E)$ be at time $t$. At a later time $t' > t$, we observe an extension of $G_t$, such that: $G_{t'} = (N \cup \Delta N, E \cup \Delta E)$ where $\Delta E \cap (N \times N) = \varnothing$. Figure 3 depicts that $G_{t'}$ is composed of $G_t$ augmented with newly added node $a3$ and its corresponding $a3 \rightarrow x2$ and $a2 \Rightarrow a3$ relations. We now can introduce $Popularity_t(x)$ for a node $x$ at time $t$. This provides the popularity of $x$ in $G_t$. [10]

*3.3. The Project-level Software Universe Graph (P-SUG)*

We consider a set of the same name nodes for a node $x$, such that: $Project(x) \equiv \{y | y \overset{+}{\Rightarrow} x \ \vee x \overset{+}{\Rightarrow} y \ \vee x = y\}$ where $a \overset{+}{\Rightarrow} b$ is the transitive closure on any update-relation $a \Rightarrow b$. The *name* attribute determines project membership. P-SUG is a merged graph of a SUG whose same name nodes are merged into a single node.[11] Consider the example in Figure 4. Figure 4(a) shows a SUG with respective projects annotated. Figure 4(b) depicts the related P-SUG.

**Definition 4.** *Variety represents the number of different projects that depend on a project release*

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

[11]Formally we define the P-SUG $G' = (N', E')$ of a SUG $G = (N, E = E_{dep} \cup E_{up})$ where $N' = \{Project(n) | n \in N\}$ *and* $E' = \{Project(a) \rightarrow Project(b) | a, b \in N \wedge a \rightarrow b \in E_{dep}\}$

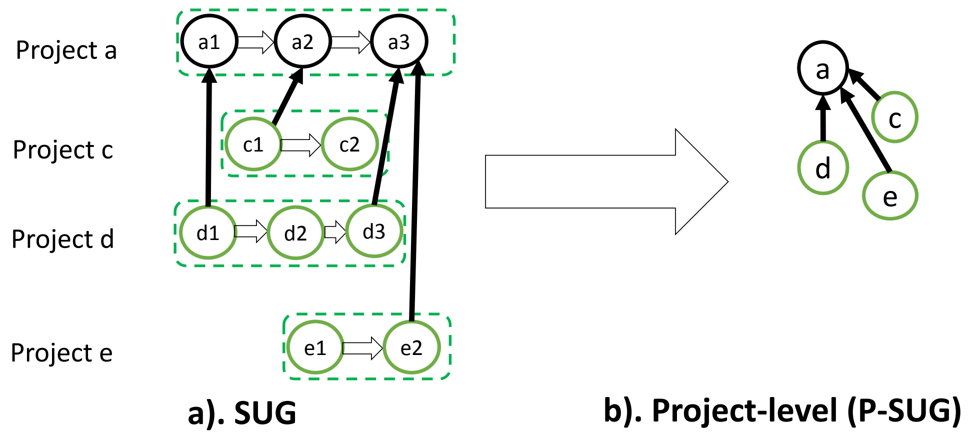


**Figure 4:** Illustrative example of a SUG merged to a P-SUG

.

Let variety for a P-SUG node $u$ be defined as:

$$variety(u) \equiv |rev\text{-}depend(u')| \tag{4}$$

where $u'$ is a merged P-SUG node from $u$. For example, in Figure 4, $variety(a1) = |\{rev\text{-}depend(a)\}| = |\{c, d, e\}| = 3$.

## 4. SUG Adoption-diffusion Profiling

*Profile types.* To demonstrate implementation of the model, we show how the SUG can be leveraged particularity for library adoption-diffusion (introduced in Section 2). As an extension on our work on Library Dependency Plots [15], we introduce Diffusion Profiling (DP). For any project releases, DP is a pair of popularity and variety at any given point in time $t$, such that $popularity_t(x)$ and $variety_t(x)$ for a SUG node $x$. For popularity, we plot the number of software projects using a particular release of the project. Conversely in the variety plot, we track the number of projects that use a specific release. The DPs provide a temporal means to evaluate popularity and the adoptive behavior nature. Since DPs plot both the $popularity_t$ and corresponding $variety_t$ on a SUG, we can use them to understand the adoption-diffusion at both project release and project levels.

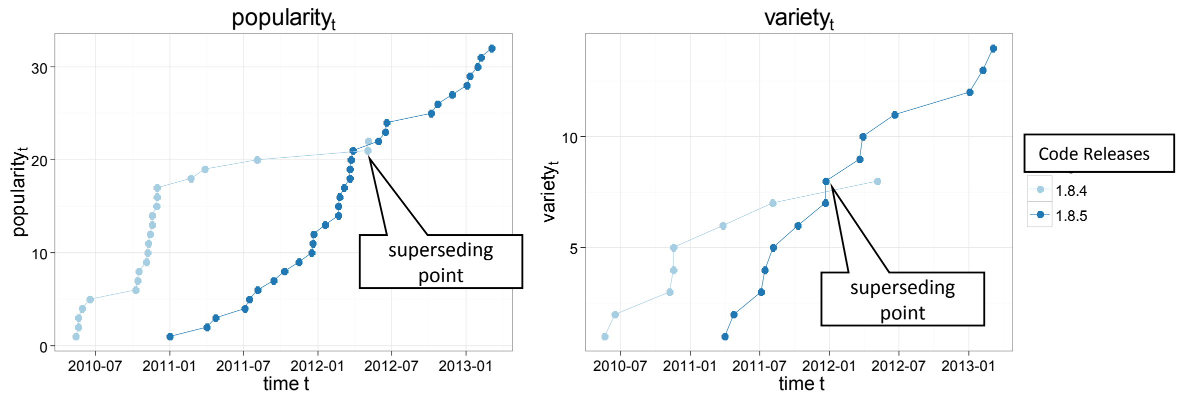


**Figure 5:** A simple example of a Diffusion Profile (DP) for Maven MOCKITO-CORE project release `1.8.4` and `1.8.5`

*Profile characteristics (Superseding point and Curve Steepness).* Particularly interesting characteristic is the temporal *superseding point* (ss point), which is the time at which the point where one project release popularity overtakes another. A very attractive version is defined as to have no other library versions superseding it. Another characteristic of the DPs is the steepness of the curve; when the curve halts, and when the curve is *superseded* by a successive release curve. A saturated curve may suggest that a release is no longer attractive and not recommended for adoption.

In Figure 5 we shows an example of the profile characteristics of the DPs of MOCKITO-CORE project. 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 $popularity_t$ `(2012-6)` and $variety_t$ `(2011-12)`. In both cases, we conclude that $mockito - core_{1.8.5}$ is recommended as the more attractive project release version to adopt.

## 5. Empirical Evaluation

### *5.1. Research Questions*

To evaluate our SUG approach we modeled dependency and evolution of releases within a real super repositories. The goal of the evaluation is to answer the following questions:

- **RQ1** *Are we able to model a real-world super repository as an SUG? and if so, what knowledge can we extract from the model?* We want to extract interesting observations such as internal reuse, that occurs within the super repository.

- **RQ2** *Can we leverage the SUG to profile library adoption-diffusion? and if so, what useful patterns are observed?* We want to use DPs to recommend practical library migrations.

## *5.2. Research Method*

For the first research question, the research method is by empirical study of a typical super repository. For the second method, through use cases, we demonstrate practicality of the DPs.

*Research Method for RQ1.* For the first research question, we first provide a detailed description of the construction of a SUG, including the node, edges and attributes definitions and statistics. To understand the reuse within a SUG, we measure how many projects are being used internally. Thus, for a SUG $U = \{N, E\}$ and its corresponding P-SUG $U' = \{N', E'\}$:

$$reuse = |\bigcup_{n \in N} rev\text{-}depend(n')| \tag{5}$$

where $n'$ is merged node in $N'$ from $n$.

*Research Method for RQ2.* For the second research question, we use two popular libraries from the built SUG in our case studies, each used to illustrate practicality of our approach. Concretely, for the *adoption-diffusion* profiling, we analyze for each library the profile types and profile characteristics (i.e, superseding point and curve steepness). We also show how each library adoption-diffusion profile may correspond to real world changes. There are many other factors to consider, however in this study, we manually cross-reference with release logs for supporting evidence to explain the different profile characteristics observed.

## *5.3. Dataset*

For RQ1, we model the Maven super repository, which hosts many JVM project artefacts. Most projects in this super repository are open-source Java, Scala or Clojure libraries (referred to as artefacts). Recently the Maven

**Table 1:** SUG statistics for Maven SUG

| | **Maven Super Repository** |
|---|---:|
| time period | 2005/11/03 – 2014/12/30 |
| # of SUG nodes | 1,664,648 |
| # of SUG floating nodes | 11,482 |
| # of P-SUG nodes(# of projects) | 37,638 |
| # of P-SUG nodes depended on by any other P-SUG nodes(# of reused projects) | 26,156 |

**Table 2:** Depend-relation edge statistics for Maven P-SUG

| | **outgoing edges** | **incoming edges** |
|---|---:|---:|
| Min | 1 | 1 |
| 1st Quartile | 3 | 2 |
| Median | 5 | 2 |
| Mean | 11.29 | 9.02 |
| 3rd Quartile | 12 | 5 |
| Max | (ORG.GLASSFISH) 1,166 | (JUNIT) 9,691 |

libraries have been gaining widespread usage through dependency management such as `maven2` and `gradle` build tools. We conducted our experiments on a local offline copy of the super repository, snapshot from 2005/11/03 to 2014/12/30. For RQ2, we employ two well-known Maven libraries for our case study. We use the SUG to analyze 6 versions (2.1 – 2.6) of COMMONS-LANG, a helper utility library and 11 versions (2.1 – 4.0) of ASM a java bytecode manipulation library. The intention is to show the different features of the adoption-diffusion plots.

All tools, scripts, data and result of systems are available from the paper's replication package at:
`http://sel.ist.osaka-u.ac.jp/people/raula-k/SUG/index.html`.

## 6. Results

### *6.1. Construction of the SUG*

For the Maven super repository, we construct the SUG from the POM configuration file. Every project in the Maven repository includes a Project

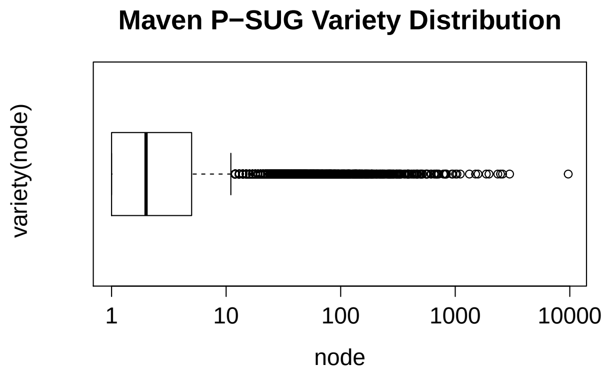


**Figure 6:** Maven variety statistics (incoming depend-relation edges)

Object Model file (i.e., `POM.xml`), that describes the project's configuration meta-data —including its compile-time dependencies[12]. We customized a tool[13] that implements the maven-model[14] parser to extract related SUG edges dependency information from all release version of the POM-files in the repository. Similarly encountered by Raemaekers[23], Maven's dependency management mechanism[15] is rather complex with elements such as transitive and imported POMs.Using the formalized model we built the Maven SUG $M$ where $M(N, E_{dep} \cup E_{up})$. Taking $x \in N$, each property is as follows:

- $E_{dep}$. The `<dependency>` attribute of the `POM.xml` explicitly references the use relation between artefacts. At this stage, we do not resolve transitive dependencies.
- $E_{up}$. The `<version>` attribute of the `POM.xml` explicitly references the release version of an artefact. Using the *time* attribute of the node, we then determine the order of nodes within a project.
- $x.name$. The `<artifactId>` was originally used, however it was found

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

[13]PomWalker: `https://github.com/raux/PomWalker`

[14]maven-model version 3.1.1. Our tool can handle Maven 1.x, 2.x and 3.

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

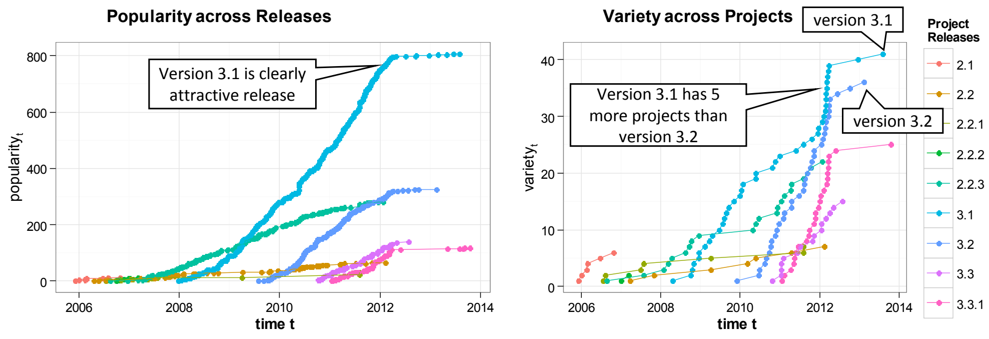
in many cases to be too generic. The concatenation of `<groupId>` produced a more unique project separation.

- *x.release*. The `<version>` attribute of the `POM.xml` explicitly references the release version of an artefact.
- *x.time*. The time-stamp of when the artifact (jar file) was uploaded into Maven was used to extract time of the node.

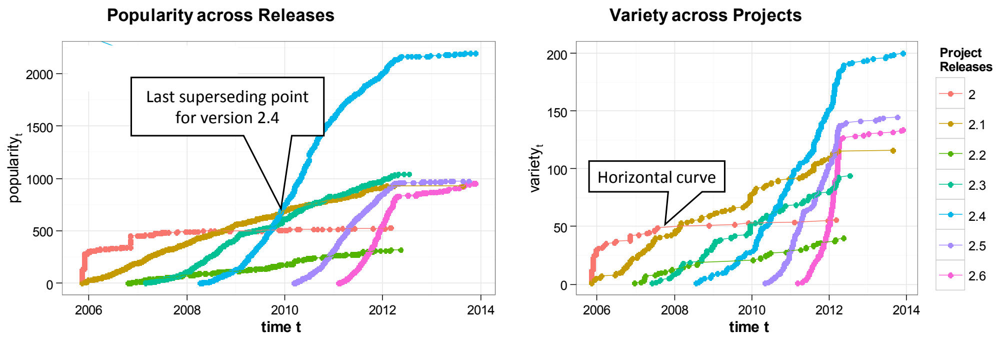
A downside of using `<groupId>` as the name attribute, is that common projects are lost if they have moved domain (i.e., changed`<groupId>`). An example is when the FINDBUGS library change groupID from
`<net.sourceforge.findbugs>` to `<com.google.code.findbugs>`. Although our tool is unable to resolve explicit references, it is able to handle inheritance attributes of Super POM. Through the `Dependency Management` attribute the parent and child POMs files were resolved.

Table 1 details the constructed Maven SUG. As shown we were able to mine and generate 1,664,648 nodes, spanning across 9 years. Independent software releases (i.e.,nodes without incoming or outgoing depend-relation edges) are refereed to as floating nodes in the SUG. We depict internal reuse within super repository (26,156 projects used by 37,638 projects). The result is typical as most Maven artifacts are known to comprise of libraries or frameworks. Statistical summary of the outgoing and incoming depend-relation in P-SUG (known as the variety) is presented in Table 2 and Figure 6. We find that ORG.GLASSFISH project is the most depending project with the most outgoing edges in this SUG. Conversely, testing library JUNIT is found to be the most depended upon library adopted across a variety of projects in the super repository.

In the Maven super repository, there is a high chance that a project may have internal dependencies (with 26,156 projects used by 37,638 projects). Very few nodes are floating with no dependencies. Most nodes have very low depend-relations edges (median of 5 outgoing and 2 incoming). However, there exists a subset of very popular projects such as JUNIT, that are popular across the super repository or ORG.GLASSFISH that depend on many other projects.



(a) ASM



(b) Commons-lang

**Figure 7:** Diffusion Plots for selected Maven projects. The left hand depicts the *popularity*$_t$ while the right shows the *variety*$_t$ for their respective releases.

## *6.2. Maven Profiling Types*

In Figure 7(a), the *popularity*$_t$ plot clearly depicts ASM$_{3.1}$ as the most attractive release. However, in the corresponding *variety*$_t$ plot, it is has only 5 additional projects than ASM$_{3.2}$. Manual analysis of the `pom.xml` file indicates many of the popularity counts belong to large frameworks like `com.sun.jersey.glassfish`. We observe these dependents have a rapid release cycle, each time creating a depend edge in the SUG. Although this popularity may indicate a library's stability within this domain, it does not mean that it is widely used elsewhere in the super repository.

*variety*$_t$ is a more reliable measure of widespread use across the super repository. Libraries with high *popularity*$_t$ but low *variety*$_t$ are possibly specialized or depended upon by projects with rapid release cycles.

### *6.3. Profile characteristic (Superseding Point)*

From Figure 7(b), the DPs depict a case where an older release is still very much used today. We observe COMMONS-LANG$_{2.4}$ (light blue) as still the most popular (no other version has passed its superseding point). This is consistent in both $popularity_t$ and $variety_t$ plots, making it the most attractive release.

*Manual cross-check with documentation.* Looking at the logs[16], we notice that COMMONS-LANG$_{2.4}$ first supported the Java JDK version 1.2. Since version 3.x (released in 2011-07-18) supports Java JDK 5.0, version 2.5 (released in 2010-2-23) or 2.6 (released in 2011-01-16) showed less adoption.

| Older releases may be still an attractive release for adoption. |
|---|

### *6.4. Profile characteristic (Curve Steepness)*

The steepness of the curve can indicate trends of popularity. For instance in the $popularity_t$ plot of Figure 7(b), we observe that COMMONS-LANG$_{2.4}$ (light blue) experienced a very strong diagonal (attractive), which is now slowing down. From 2012 onwards, all curves become horizontal, indicating all releases are not attractive for adoption.

*Manual cross-check with documentation.* As noted in the documentation[17], the next release (COMMONS-LANG3) version 3.0 was moved to a new project, allowing both projects to be used side-by-side. The horizontal curve may be caused by this competing version 3.0.

| Diagonal or vertical spikes shows potential attraction by the crowd while horizontal curves indicate lesser dependents than before. |
|---|

Based on the results, we return to practical recommendations for our case study. COMMONS-LANG$_{2.4}$ is a viable library to adopt or continue use as it is very popular and the curve steepness shows that the crowd still find the version attractive. However, the horizontal curve at the end of all releases suggest that system maintainers should consider (COMMONS-LANG3) version 3.0 to be used-side-by-side with COMMONS-LANG$_{2.4}$.

[16]`http://commons.apache.org/proper/commons-lang/release-history.html`

[17]`http://commons.apache.org/proper/commons-lang/article3_0.html`

For the ASM library, although $\text{ASM}_{3.1}$ has more popularity appeal overall, the $variety_t$ plot suggest $\text{ASM}_{3.2}$ is just as attractive for adoption. To decide between the two libraries, the system maintainer can not seek documentation[18] on new bug fixes and features for $\text{ASM}_{3.2}$.

Finally, we address the research questions. For RQ1, we were able to successfully generate the SUG for the Maven Super Repository. For RQ2, we demonstrated how diffusion patterns can be important indicators for potential library candidate adoptions.

## 7. Discussion

### *7.1. Study Implications*

Advancements in data storage and mining repository techniques and tools, make possible the study of popularity and 'follow the crowd' approaches. We have shown that the SUG can model the update and depend relationships in a typical super repository.

As mentioned in Section 1, there are many different factors for updating libraries: technical (vulnerabilities), organizational (platform specification or use of a component) and personal (knowledge) that influences the decision to migrate libraries. In the results, we show that the SUG metrics and visualizations provide a starting point of investigation of the crowd 'consented' libraries to adopt. It is a much easier start before browsing all the release logs to differentiate differences. Other methods such as library compatibility are more precise but requires compilations and running of the tool.

There is related work that focus on popularity such as [24] and recently Hora et. al [25]. However, we show in the $popularity_t$ and the corresponding $variety_t$ DPs similar popularity trends.

In this paper, we only leverage the SUG metrics to describe adoption-diffusion. We believe that the model can generate other more complex results such as co-dependency evolution and recommendation of outside libraries or similar libraries. The generic and robust nature of the SUG makes for promising future extensions.

### *7.2. SUG Extensions*

Our SUG model is designed to rely on the dependency chains but differs from typical graph cyclic based approaches such as ranking (such as page

[18]`http://asm.ow2.org/history.html`

ranking), reference counting and component ranking, which is common for measuring popularity [26], [27]. The current graph modeled structure allows for faster and scalable querying, which we utilize for the adoption-diffusion.

We envision the SUG as a foundation in which many other features can be built. As we study more systems, we will consider integrating 'containment' and 'transitive' concepts of object-oriented software into the SUG. We also plan to address issues of authentication of the `name` attribute, and to expand beyond the name attribute for project classifications, by incorporating more sophisticated techniques and tools used in 'code clone' such as code clone detection [28], [29] and 'origin' analysis [30], [31] to determine a common project. Another complex but useful operation that was not presented in this paper is the tracing of systems that have abandoned or dropped a library dependency.

### *7.3. Threats to Validity*

The main threat to the internal validity is the real-world evaluation by maintainers. We have been working closely with system integration industrial partners to develop and test our visualizations. We argue though our examples are sufficient to demonstrate possible library recommendations. In this study, we used the pom.xml attributes to build the SUG. The abstract nature of the SUG allows for incorporation of other programming languages which provide their own library hosting repository. Therefore, we believe the SUG to be a universal approach for any type of super repository.

For external threats, our datasets only includes information about dependencies that are explicitly stated in project configuration files, such as the Maven `POM` configuration files. we assume consistencies between the stated dependencies and actual dependencies. Also, we do not take into account unaccounted reuse such as copy-and-paste and clone-and-own from within the libraries. 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.

## 8. Related Work

### *8.1. Popularity Metrics*

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 [32]. Eisenberg et al. improve navigation through a library's structure using the popularity of its elements to scale their depiction [33]. De Roover et al. explore library popularity in terms of source-level usage patterns [34]. Popularity over time has received less attention. Mileva et al. study popularity over time to identify the most commonly used library versions [24]. Follow-up work applies the theory of diffusion to identify and predict version usage trends [35]. Similar to our diffusion work, Bloemen et al. [36] explored the diffusion of Gentoo packages. Using the Bass Diffusion Model, they modeled the diffusion of Gentoo packages over time. Other related work includes the 'library migration graphs' of Teyton et al.[26]. Recently Hora introduced apiwave in visualizations to show popularity trends at the API level. [25].

Our work extends on popularity for more indepth analysis of the 'wisdom of the crowd'. Our study investigates co-dependency and diffusion instead of migration. Consequently, our graph implements an incremental approach as opposed to the cyclic migration graph model.

### *8.2. The Software Repository Universe as Ecosystems*

Recently, there has been an increase in research that perceives software systems as ecosystems. Work such as Bosch [37] have studied the transition from Product Lines to an Software Ecosystem approach. German et al. [21] studied the GNU R project as an ecosystem over time. Since the projects inception, the studied found that user-contributed systems have been growing faster than core-systems and identified differences of how they attracted active communities. Mens et al [38] perform ecological studies of open source software ecosystems with similar results.

Haenni et al. [22] 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.

### *8.3. Code Search and Library Recommendation Systems*

Code search is prominent among research on software reuse with many benefits for system maintainers [39]. Examples of available code search engines are google code[19] and black duck open hub [20]. Tools such as Ichi-tracker [40], Spars [27], MUDAblue [41] and ParserWeb [42] 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 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 co-dependency and adoption-diffusion behavior.

Most existing library recommendation work are based on commonly used together patterns at the method level, i.e., API usage patterns at the method level of granularity. Other related work only recommend support for existing libraries in systems, using code examples or linkage to online learning resources. The most related work of recommendation at the library level of granularity is by Thung et al. [43]. Through Mining Software Repositories (MSR), they use association rule mining on historic software artifacts to determine commonly used libraries. Inspired by these existing work, we believe the SUG model can be leveraged by to expand the current work and provide a means towards better library recommendation systems.

In regard to the SUG attributes and properties, there exists many related definitions of software variability and dependency relationships. In Software Product Line Engineering (SPL), terms such as 'product' variability has been used extensively [37], [44], [45]. In the code clones field, Kim et al. [46] coined clone 'genealogies' to track variability between software of similar origins. In addition, systems and libraries are not explicitly distinguished. The co-dependency operations on the SUG demonstrate more 'basic' 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 or dropped a library dependency.

## 9. Conclusion and Future Work

OSS libraries are now prominent in modern software development. With the rise of super repositories such as Maven, Sourceforge, and GitHub,

[19]`https://code.google.com/`

[20]`https://code.openhub.net/`

several opportunities have arisen to uncover insights valuable to the management of library dependencies through intelligent super repository mining. In this paper, we present the SUG model as a means to represent, query and quantify different super repositories in a generic manner. Immediate future work focuses on evaluating the SUG with actual system maintainers. We are also developing SUGs prototypes of different super repositories to gain feedback and explore other potential future uses of the model.

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.

## Acknowledgments

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] L. Hainemann, F. Deissenboeck, M. Gleirscher, B. Hummel, M. Irlbeck, On the extent and nature of software reuse in open source java projects, in: Proc. of 12th Intl. Conf. on Top Prod. Soft. Reuse, 2011, pp. 207–222.

[3] Spring io, accessed 2015-08-01, `https://spring.io/`, 2015.

[4] Apache commons, accessed 2015-08-01, `http://commons.apache.org/`, 2015.

[5] The maven central super repository, accessed 2015-08-01, `http://search.maven.org/`, 2015.

[6] Sourceforge super repository, accessed 2015-08-01, `http://sourceforge.net/`, 2015.

[7] Github super repository, accessed 2015-08-01, `https://github.com/`, 2015.

[8] 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).

[9] M. Mattsson, J. Bosch, M. E. Fayad, Framework integration problems, causes, solutions, Communications of the ACM 42 (1999) 80–87.

[10] M. E. Fayad, D. C. Schmidt, R. E. Johnson, Building Application Frameworks: Object-oriented Foundations of Framework Design, John Wiley & Sons, Inc., New York, NY, USA, 1999.

[11] C. Bogart, C. Kästner, J. Herbsleb, When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies, in: Proceedings of the ASE Workshop on Software Support for Collaborative and Global Software Engineering (SCGSE), 2015.

[12] R. G. Kula, D. M. German, T. Ishio, K. Inoue, Trusting a library: A study of the latency to adopt the latest maven release, in: 22nd IEEE Intl. Conf. on Soft. Anal., Evol., and Reeng., (SANER2015), Montreal, Canada, March 2-6, 2015.

[13] B. Dagenais, M. P. Robillard, Semdiff: Analysis and recommendation support for api evolution, in: Procc of the Intl Conf. on Soft. Eng., (ICSE2009), Washington, DC, USA, 2009, pp. 599–602.

[14] Clirr tool and library, accessed 2015-08-01, `http://clirr.sourceforge.net/`, 2015.

[15] R. G. Kula, C. D. Roover, D. M. German, T. Ishio, K. Inoue, Visualizing the evolution of systems and their library dependencies, Proc. of IEEE Work. Conf. on Soft. Viz. (VISSOFT2014) (2014).

[16] Y. Yano, R. G. Kula, T. Ishio, K. Inoue, Verxcombo: An interactive data visualization of popular library version combinations, in: Proc. of Intl Conf. on Prog. Comp., (ICPC 2015), Firenze, Italy, May 18-19, 2015.

[17] E. M. Rogers, Diffusion of innovations, 5th ed., Free Press, NY, 2003.

[18] S. Chuan-Fong, V. Alladi, Beyond adoption: Development and application of a use-diffusion model, 2004, pp. 59–72.

[19] S. K. Sowe, L. Angelis, I. Stamelos, Y. Manolopoulos, Using repository of repositories (rors) to study the growth of f/oss projects: A meta-analysis research approach, in: IFIP Intl Fed. for Info. Proces., 2007.

[20] M. Lungu, M. Lanza, T. Grba, R. Heeck, Reverse engineering super-repositories, in: Work. Conf, Rev. Eng. (WCRE2007), 2007.

[21] D. M. German, B. Adams, A. E. Hassan, The evolution of the r software ecosystem, Proc. of European Conf. on Soft. Main. and Reeng. (CSMR2013) (2013) 243–252.

[22] N. Haenni, M. Lungu, N. Schwarz, O. Nierstrasz, Categorizing developer information needs in software ecosystems, in: Proc. of Int. Work. on Soft. Eco. Arch. (WEA13), 2013, pp. 1–5.

[23] S. Raemaekers, A. van Deursen, J. Visser, Measuring software library stability through historical version analysis, in: Proc. of Intl. Comf. Soft. Main. (ICSM), 2012, pp. 378–387.

[24] Y. M. Mileva, V. Dallmeier, M. Burger, A. Zeller, Mining trends of library usage, in: ERCIM Workshops, 2009, pp. 57–62.

[25] A. Hora, M. T. Valente, apiwave: Keeping track of api popularity and migration, in: Intl Conf. on Soft. Main. and Evol., (ICSME2015), 2015.

[26] C. Teyton, J.-R. Falleri, X. Blanc, Mining library migration graphs, in: Proc. of. Work. Conf. on Rev. Eng. WCRE2012, 2012, pp. 289–298.

[27] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, S. Kusumoto, Ranking significance of software components based on use relations, Software Engineering, IEEE Trans. 31 (2005) 213–225.

[28] C. K. Roy, J. R. Cordy, A survey on software clone detection research, in: Technical Report No. 2007-541,Queens University, Canada, 2007.

[29] T. Kamiya, S. Kusumoto, K. Inoue, CCFinder: a multilinguistic token-based code clone detection system for large scale source code, IEEE Trans. on Soft. Eng. 28 (2002) 654–670.

[30] M. Godfrey, L. Zou, Using origin analysis to detect merging and splitting of source code entities, volume 31, 2005, pp. 166–181.

[31] J. Davies, D. M. German, M. W. Godfrey, A. Hindle, Software bertillonage: Finding the provenance of an entity, in: Proc. of Work. Conf. on Min. Soft. Repo., 2011, pp. 183–192.

[32] R. Holmes, R. J. Walker, Informing Eclipse API production and consumption, in: OOPSLA2007, 2007, pp. 70–74.

[33] D. S. Eisenberg, J. Stylos, A. Faulring, B. A. Myers, Using association metrics to help users navigate API documentation, in: VL/HCC2010, 2010, pp. 23–30.

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

[35] Y. M. Mileva, V. Dallmeier, A. Zeller, Mining API popularity, in: TAIC PART, 2010, pp. 173–180.

[36] R. Bloemen, C. Amrit, S. Kuhlmann, 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.

[37] J. Bosch, From software product lines to software ecosystems, in: Proc.of the Int Soft. Prod. Line (SPLC '09), 2009, pp. 111–119.

[38] T. Mens, M. Claes, P. Grosjean, Ecos: Ecological studies of open source software ecosystems, in: Soft. Main. Reeng. and Rev. Eng. (CSMR-WCRE), 2014, pp. 403–406.

[39] S. Bajracharya, A. Kuhn, Y. Ye, Proc. of work. on search-driven dev.: Users, infrastructure, tools, and evaluation (suite 2011), in: Proc. of the Intl Conf. on Soft. Eng., 2011.

[40] K. Inoue, Y. Sasaki, P. Xia, 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., (ICSE2012), IEEE Press, Piscataway, NJ, USA, 2012, pp. 331–341.

[41] S. Kawaguchi, P. K. Garg, M. Matsushita, K. Inoue, MUDABlue: an automatic categorization system for open source repositories, Journal of Systems and Software 79 (2006) 939–953.

[42] S. Thummalapenta, T. Xie, Parseweb: A programmer assistant for reusing open source code on the web, in: Proceedings of the IEEE/ACM Intl. Conf on ASE, ASE '07, ACM, New York, NY, USA, 2007, pp. 204–213.

[43] F. Thung, D. Lo, J. Lawall, Automated library recommendation, in: Proc. of Work. Conf. on Rev. Eng., (WCRE2013),, 2013, pp. 182–191.

[44] C. Seidl, 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.

[45] M. Nonaka, K. Sakuraba, K. Funakoshi, A preliminary analysis on corrective maintenance for an embedded software product family, IPSJ SIG Technical Report 2009-SE-166 (2009) 1–8.

[46] M. Kim, V. Sazawal, D. Notkin, G. Murphy, An empirical study of code clone genealogies, in: Proc. of Euro. Soft. Eng. Conf. with Symp. on Found. of Soft. Eng., 2005, pp. 187–196.