Mining Library Migration Graphs

Cédric Teyton, Jean-Rémy Falleri, Xavier Blanc
Univ. Bordeaux, LaBRI, UMR 5800
F-33400 Talence, France
Email: name.surname@labri.fr

***Abstract*—Software systems intensively depend on external libraries, chosen at conception time. However, relevance of any library irremediably changes during projects and/or library life cycle. As a consequence, projects developers must periodically reconsider the libraries they depend on, and must think about *library migration*. When they want to migrate their libraries, they then have to identify candidate libraries that offer similar facilities and thus can substitute to each other. They also have to compare candidates to choose the one that best fits their needs. Finding a relevant library replacement is a well known tedious and time-consuming task. In this paper, we propose an approach that identifies sets of similar libraries and that produces what we call *library migration graphs* that show how existing projects have performed migrations among them. These graphs, constructed from the observation of a large number of software projects, ease the discovery and selection of library replacements.**

I. Introduction

Almost all software projects depend on external libraries that provide useful technical facilities. Examples of such libraries are *JUnit* for unit testing or *Log4J* for logging. The relevance of a library for a given project irremediably changes during the project and/or the library life cycle. As a consequence, projects developers must periodically reconsider the libraries they depend on, and must think about library migration when the libraries they depend on are not updated, or when competing ones appear with more features or better performances for instance.

Even if library migrations are not so frequent in project life cycle, we observed that they do exist. By looking at the source code repository of several thousands of open-source software projects, the migrations we observed fall into the following categories:

- ***Convenience***, as expressed by these developers: "*changed from MySQL to HSQLDB, so that it can be built on any machine*"[1] and "*changed JTA dependency to use geronimo spec to avoid having to download and install JTA manually*"[2].
- ***Outdated***: "*switched to using Google Guava library instead of older collections library*"[3].
- ***Incompatibilities***: "*Added Slf4j in favor of Commons-logging to avoid conflict with Hibernate Validator*"[4] or "*port logging to slf4j (commons-logging has classloader issues)*"[5]

Whatever the category, developers that are looking for replacement libraries can currently only use general purpose search engine, such as Google. For example, developers that want to migrate from the `Commons-logging` library would probably query Google with "*logging library Java*". Such a query returns many technical websites that provide partial, incomplete and out-of-date answers of libraries to migrate to. Therefore, to not end up in a tedious and time-consuming process, there is a need for an approach that helps to find adequate candidate libraries to migrate to, which is the purpose of this paper.

Library migration raises two main issues. The first one consists in identifying libraries that offer similar facilities and thus can substitute to each other. The second one consists to provide comparison criteria to help developers to choose the ideal library to migrate to. For example, if a project wants to migrate from *Log4J*, it may consider *SLF4J* or *Commons Logging* as they both address logging, and *SLF4J* may be preferred as many more projects have chosen *SLF4J* rather that *Commons-Logging*.

In this paper, we propose to address these two issues by analysing large corpus of software projects to see how their dependencies evolve in order to exhibit common behaviours. We assume that if we observe that a large number of projects migrate from *Log4J* to *SLF4J*, it is then relevant for every project using *Log4J* to consider *SLF4J* as a good candidate to migrate to. We base our analysis on modern software project management tools that ease specification and maintenance of software dependencies. Whatever the tool, Maven[6], Apache Ivy[7], or Gradle[8] for instance, their principles are quite the same. Each project defines its dependencies with other projects in a configuration file. The tool then checks the dependencies during the life cycle of the project (compilation, testing, deployment, etc.). Thanks to such configuration files, which make dependencies explicit, we can analyse how library migrations are performed by projects, with the intent to exhibit common migration rules. Such an analysis is even more simple when the build management tools propose a central repository that stores all configuration files of all projects, which is the case at least for Maven.

Our contribution is twofold. First, we propose a mining approach that exhibits migration rules and that can be applied

[1]http://code.google.com/p/jlibs/source/detail?r=955
[2]http://code.google.com/p/mybatis/source/detail?r=1996
[3]http://code.google.com/p/google-gson/source/detail?r=619
[4]http://code.google.com/p/sventon/source/detail?r=1681
[5]http://code.google.com/p/dyuproject/source/detail?r=668

[6]http://maven.apache.org/
[7]http://ant.apache.org/ivy/
[8]http://www.gradle.org/

to any tool that manages library dependencies. Our approach computes what we call *migration graphs* that exhibits categories of similar libraries with relationships representing how often migrations have been performed between them. Second, we present the result of our mining approach applied to the Apache Maven[9] project management system. This result presents several categories such as logging, data base and XML analysis. We have compared our migration graphs with usage trends of libraries in order to show that they add valuable information.

The remainder of this paper is structured as follows. Section II explains our process to mine library dependencies in order to generate library migration graphs. Section III presents a case study of our approach on the projects located on the major Maven repository. Section IV exposes the migration graphs extracted during the case study. Section V discusses the limitations of our approach. Section VI presents the related work, while Section VII uncovers the future work and concludes.

II. Approach

In this section, we first present the abstract model we define to represent evolving software projects. Based on this model, we present the algorithm we use to extract what we call *migration rules* that abstract library migrations performed by projects. Finally, we present what we call migration graphs that express categories of similar libraries and that highlight library migration flow.

A. Dependency Model

In order to be independent of any project management tool, we propose to abstract the data needed to perform our analysis in what we call a dependency model. This model is very simple as it only contains the set of analysed projects, their list of revisions and, for each revision the associated set of dependencies. More formally, P is the set of software projects of the model (we consider that a library is a project too). Each project $p \in P$ has an associated totally ordered set R_p of project revisions. This set is sorted chronologically according to the revision dates. For a project revision $r \in R_p$, we define $d(r) : R_p \to \mathcal{P}(P)$ the set of its dependencies.

Let us illustrate our model with an example. We assume six projects, two development projects ($\mathtt{P_A}$ and $\mathtt{P_B}$) and four libraries (JUnit, JMock, Log4J and SLF4J). Table I presents this dependency model with revisions of $\mathtt{P_A} : (\mathtt{P_A^1}, \mathtt{P_A^2}, \mathtt{P_A^3})$ and of $\mathtt{P_B} : (\mathtt{P_B^1}, \mathtt{P_B^2})$, associated with their corresponding dependencies.

Based on this model, for a given revision r_i, we introduce $\mathrm{rem}(r_i) = d(r_{i-1}) \setminus d(r_i)$ and $\mathrm{add}(r_i) = d(r_i) \setminus d(r_{i-1})$ that give respectively the added and removed dependencies at a given revision with respect to the previous revision. With our example, we can see that $\mathrm{rem}(\mathtt{P_A^3}) = \{\mathtt{Log4J}\}$ and $\mathrm{add}(\mathtt{P_A^3}) = \{\mathtt{SLF4J}\}$. These two sets will help us to identify library migrations performed at a given revision.

[9]http://maven.apache.org/

Project	Revisions / Dependencies		
$\mathtt{P_A}$	$\mathtt{P_A^1}$ {JUnit}	$\mathtt{P_A^2}$ {JMock,Log4J}	$\mathtt{P_A^3}$ {JMock,SLF4J}
$\mathtt{P_B}$	$\mathtt{P_B^1}$ {JMock}	$\mathtt{P_B^2}$ {JUnit}	

TABLE I: Example of projects with their revisions and their dependencies

B. Mining library migration rules

A project performs a library migration when it exchanges one of its dependent library by another one. Such migration includes two libraries with different names, and thus discards library upgrades. Therefore, if a project updates a dependency from *JUnit 3.8* to *JUnit 4.8*, this is not considered as a migration in our approach. More formally, we define a library migration rule $m \in M$ as a couple $(s, t) \in P^2$ with $s \neq t$. s is the old library (source) and t is the new library (target). A migration rule denotes the removal of s in favour of t. Note that we assume that migration rules are one to one rules. We do not consider cases where one or several libraries are replaced by one or more libraries. Thus, *n:m* migration rules are not discussed in this paper.

To extract migration rules, we apply a straightforward algorithm that iterates on revisions of a set of software projects (see Algorithm 1). For each revision, our algorithm looks at the dependencies that have been removed and added. It then creates a candidate migration rule for each element of the Cartesian product of the removed and added dependencies. The removed (resp. added) dependencies are the source (resp. target) of the candidate migration rules.

Once candidate migration rules have been created, we filter out candidates that are not relevant according to a scoring function. To that extent, we define the function $r(m) : M \to \mathcal{P}(\bigcup_{p \in P} R_p)$, that returns for a given rule m the set of revisions that target the rule. We introduce the confidence $\mathrm{conf}(m) : M \to \mathbb{R}$ as the minimum value between $\frac{|r(m)|}{|\{(s,x) \in M\}|}$ and $\frac{|r(m)|}{|\{(x,t) \in M\}|}$, where $m = (s, t)$, where (s, x) with $x \in P$ that denotes any rule $r \in M$ having s as a source, and where (x, t) that denotes any rule $r \in M$ having t as a target.

With our example, the Cartesian product returns the rules $m_1 = (\mathtt{JUnit,JMock})$, $m_2 = (\mathtt{JUnit,Log4J})$, $m_3 = (\mathtt{Log4J,SLF4J})$ and $m_4 = (\mathtt{JMock,JUnit})$. Moreover $r(m_1) = \{\mathtt{P_A^2}\}$, $r(m_2) = \{\mathtt{P_A^2}\}$, $r(m_3) = \{\mathtt{P_A^3}\}$ and $r(m_4) = \{\mathtt{P_B^2}\}$. Since there are two rules having JUnit as source and one rule having JMock as target, $\mathrm{conf}(m_1) = \min(\frac{1}{2}, \frac{1}{1}) = 0.5$. Similarly, $\mathrm{conf}(m_2) = 0.5$, $\mathrm{conf}(m_3) = 1$ and $\mathrm{conf}(m_4) = 1$. If we set a threshold of confidence of 1, only the rules m_3 and m_4 are filtered in.

As our filter may return false positives, candidate rules are manually checked. The goal of this manual check is to build the set E of expert rules, which are the rules that correspond to correct migrations. To help this manual step, we propose a pseudo automatic analysis of description mes-

Algorithm 1 Our migration rules mining algorithm

```
for all $p \in P$ do
    for $i = 2 \rightarrow size(R_p)$ do
        $r_i = R_p[i]$
        $R = \mathrm{rem}(r_i)$
        $A = \mathrm{add}(r_i)$
        for all $r \in R$ do
            for all $a \in A$ do
                $m \leftarrow (r, a)$
                $M \leftarrow M \cup m$
                $r(m) \leftarrow r(m) \cup r_i$
            end for
        end for
    end for
end for
```

sages attached to commit performed while a library migration occurs. Looking at such messages may provide confidence to the truth of migration rules. For instance, a commit message associated to revision $a3$ saying that the library `Log4J` has been replaced by `SLF4J` provides confidence for the migration rule m_3 =(`Log4J`,`SLF4J`). In our example, we consider that both m_3 and m_4 have been confirmed to be correct ($E = \{m_3, m_4\}$).

As a last step, we add to E the rules from M that have been incorrectly filtered out, but can be deduced from the rules of E by applying transitivity and symmetry. This step, called rules augmentation, is described in Algorithm 2. On our example, Algorithm 2 adds (`JUnit`,`JMock`) in E. Finally $E = \{m_1, m_3, m_4\}$.

Algorithm 2 Rules augmentation

```
for all $m = (s, t) \in M \setminus E$ do
    if $((s, y) \in E \vee (y, s) \in E, y \neq s)$ and $((y, t) \in E \vee (t, y) \in E)$ then
        $E = E \cup m$
    else if $((t, y) \in E \vee (y, t) \in E, z \neq t)$ and $((z, s) \in E \vee (s, z) \in E)$ then
        $E = E \cup m$
    else if $(t, s) \in E$ then
        $E = E \cup m$
    end if
end for
```

C. Building migration graphs

After having identified migration rules, we compute what we call a migration graph. The nodes of this graph are projects that have been either source or target of at least one migration rule. Nodes of a migration graph are then library projects that have been migrated. A directed arc exists between two nodes if there is at least a migration rule between the two nodes. To indicate the flow of migration between the different libraries, the arcs are labelled using $r(m)$ (an arc $m = (s, t)$ is labelled with $|r(m)|$).

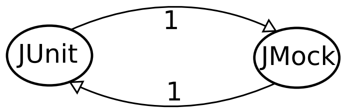


Fig. 1: A category of our example

To extract categories of similar libraries, we compute the connected components on the migration graph. Each connected component is a category, whose software projects of the category are the union of the projects contained in the connected component. An example of category is shown in Figure 1. This category shows that `JUnit` and `JMock` are similar and that projects have performed migration in the two sides.

III. Case study : Maven Central Repository

For practical reasons, we have realized our case study on projects managed under Maven because 1) it has a centralized repository system of almost 40 000 projects 2) there is a web-based REST API enabling to easily retrieve information from this repository.

A. Extracting dependencies from Maven

Maven is a build and dependencies management system for Java software projects. Each project managed by Maven requires a so-called POM (Project Object Model) XML file that contains three mandatory tags: a *groupId* that defines the group owning the project, an *artifactId* that is the unique name of the project under the *groupId*, and a *version* number. POM file also contains a *dependencies* tag where a set of dependencies can be specified, each one by the *groupId* of the dependency, its *artifactId* and optionally its *version*. Figure 2 shows an excerpt of the POM file of `JUnit` in its *4.10* version.

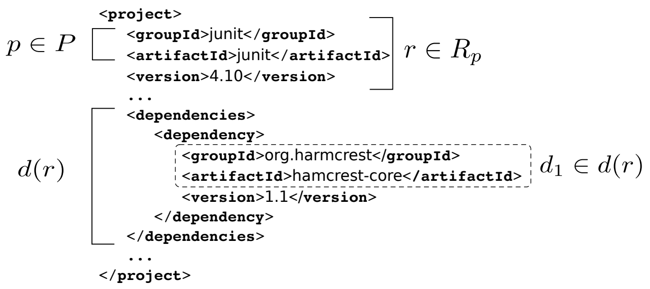
Project dependencies are stored on a Maven repository. These dependencies are Maven projects as well, and have therefore a POM file as well. The Maven Central Repository[10] (MCR) is the official Maven repository. It serves over 70 millions downloads every week[11]. It is a relevant data source for our case study for two reasons: 1) the set of projects located on the MCR is large and 2) the dependencies are specified with consistency and can be easily retrieved across projects, which allows us to concentrate on extracting the migration rules.

We now detail how we instantiate our model from the data present on the MCR. First of all, P is the set of projects located on the MCR. A project $p \in P$ is identified as an element *groupId:artifactId* according to the tags included in its POM file. Each project p has an associated set R_p containing all its versions. This set is ordered using the versioning scheme used by Maven[12] applied on the tag. To obtain a list of dependencies $d(r), r \in R_p$, we investigate the *dependencies* section of the POM file. It contains several *dependency* tags, from which we extract the *groupId* and *artifactId* tags. This process is illustrated in Figure 2.

[10]http://search.maven.org/

[11]http://www.sonatype.com/people/2010/12/now-available-central-download-statistics-for-oss-projects/

[12]http://docs.codehaus.org/display/MAVEN/Versioning



```
<project>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.10</version>
  ...
  <dependencies>
    <dependency>
      <groupId>org.harmcrest</groupId>
      <artifactId>hamcrest-core</artifactId>
      <version>1.1</version>
    </dependency>
  </dependencies>
  ...
</project>
```

Fig. 2: An excerpt of JUnit 4.10 POM file

B. Filters to improve rule generation

Within Maven, a same project can exist under several identities, whereas it should be perceived as a unique one. For example, one project can make use of either `junit` or `junit-dep`, with both referring to the `junit` project. In our opinion, if a project updates its dependency `junit` to `junit-dep`, it is not relevant to consider this as a migration rule. Also, the project `hsqldb` can be found under both *groupId* `org.hsqldb` and `hsqldb`. If a project decides to change only the *groupId* of a dependency, we should not consider it as a migration. This situation can lead to the case where two different rules exist but with a similar semantic. For example, `(org.hsqldb:hsqldb,com.h2database:h2)` and `(hsqldb:hsqldb,com.h2database:h2)` should be merged as a unique rule.

To tackle these issues, we propose a four-step process to finally produce migration rule :

1) We apply a post-processing step after the computation of the sets $\mathrm{rem}(r_i)$ and $\mathrm{add}(r_i)$. Its objective is to remove from these sets the projects that fall into the previous situations. Our solution is to consider only the *artifactId* of the projects from $\mathrm{rem}(r_i)$ and $\mathrm{add}(r_i)$, regardless the *groupId*.
2) For each element p that both belongs to $\mathrm{rem}(r_i)$ and $\mathrm{add}(r_i)$, we remove p from those two sets. From now, projects dependencies are only seen with their *artifactId*.
3) We detect the similar remaining elements from these two sets that must be discarded from the analysis. To that purpose, we compute T_p the set of tokens appearing in the *artifactId* of a project p. For instance $T_p = \{$`junit,dep`$\}$ for *junit-dep*. The segmentation is done using non-alphanumeric characters and CamelCase. Then p and p' are removed from resp. $\mathrm{rem}(r_i)$ and $\mathrm{add}(r_i)$ iff $p \in \mathrm{rem}(r_i)$ and $p' \in \mathrm{add}(r_i)$ with $T_p \cap T_{p'} \neq \emptyset$.
4) The Cartesian product can now compute the migration rules.

An example of this straightforward process is detailed in Figure 3. This example highlights the removal of misleading migration rules. For instance, thanks to our process, the rule `(junit:junit-dep)` is not returned.

As a last point, we introduce $|g(m)|$ as the set of distinct *groupIds* of projects on which $m \in M$ has been observed. This value intervenes as it may happen that several sub-projects within a similar *groupId* perform the same migration. This can be due to a lack of modularity in the project organization. It results that, if a migration rule is observed in 10 sub-projects simultaneously, its score is increased of 10 as well. However, we think that such situation bias the actual score of a migration rule. We therefore decide to compute the value $|g(m)|$ since it is a better indicator than $|r(m)|$ to estimate the popularity of a migration rule. For instance if $r(m) = \{\mathsf{g}_1{:}\mathsf{a}_1, \mathsf{g}_1{:}\mathsf{b}_2, \mathsf{g}_2{:}\mathsf{c}_1\}$, $g(m) = \{\mathsf{g}_1, \mathsf{g}_2\}$. In the remainder of the article, we use $|g(m)|$ instead of $|r(m)|$ to evaluate the popularity of a migration rule.

C. Results

The MCR provides a Web REST API to ease the access to the projects it contains. We have used this API to set-up an automatic process and to collect 38588 projects and 310571 projects releases. Looking for every $p \in P$ and their associated R_p on the repository took about 2 hours. Mining every associated POM file takes about 35 hours. Computing the rules from every successive couple of set of dependencies $d(r1), d(r2)$ takes 2 minutes. Before filtering, we obtain a list of 925 migration rules that constitute the set M.

Among the generated 925 rules, we need to filter the incorrect rules as much as possible to reduce the manual work when building the expert rules E that we finally deliver. To that purpose, we first use the previously introduced $g(m)$. We include in E only rules having a value $|g(m)| \geq 2$. The idea behind this statement is that a migration rule gains confidence if it has been observed on several unrelated projects, here at least 2. This filter builds a subset $M' \in M$ of 385 rules.

We decide to retain only rules that have a sufficient confidence value *conf(m)*. We therefore introduce a threshold of confidence value t_c that must be fixed. In order to optimize the threshold t_c, we run our tool against the previously computed set M' with different values for t_c. Then we manually rate the migration rules found, either as correct or wrong. We aim at maximizing the number of true positives while reducing as much as possible the number of false positives. Results are reported in Figure 4. Note that manually reviewing a rule is usually an easy task as it only requires to look at the websites of the projects to see if they target the same domain. Nevertheless, it is possible that a few false positive were not removed due to a misunderstanding of the libraries features. The whole task of removing the false positives took about 2 hours for rating the 385 rules.

According to the results reported in Figure 4, we choose to set t_c to 0.06 so that we achieve a precision of 0.67, containing 57 correct rules and 27 incorrect rules. Using the 57 correct rules, we apply the rule augmentation technique described in Algorithm 2. This technique managed to extend E from 57 to 80 rules. This set of 80 rules is the definitive set of expert migration rules E. Table II shows the 15 most observed migrations. The whole set E is available online[13].

[13]http://www.labri.fr/perso/cteyton/index.php?page_name=rules

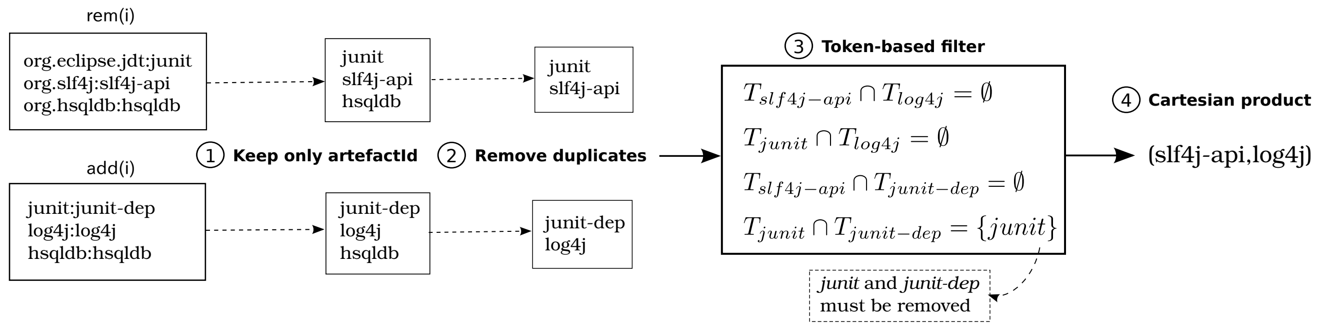


Fig. 3: Maven-specific filter to improve rule generation. From two sets of 3 dependencies, only one rule is kept after the cartesian product.

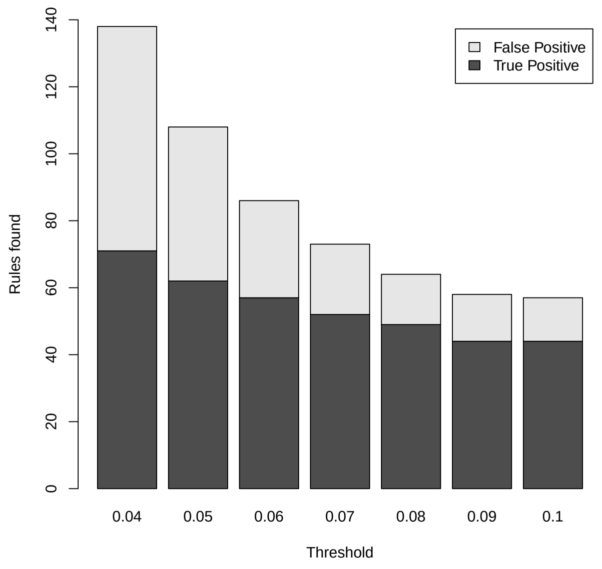


Fig. 4: Precision evaluation on t_c variations

Rule		
Source	Target	$\lvert g(m)\rvert$
commons-logging	slf4j-api	44
commons-logging	slf4j-log4j12	21
junit	testng	20
google-collections	guava	16
servletapi	servlet-api	13
testng	junit	13
slf4j-log4j12	logback-classic	13
log4j	slf4j-api	11
log4j	logback-classic	11
xbean	xmlbeans	10
commons-logging	slf4j-simple	8
hsqldb	h2	6
gfprobe-provider-client	management-api	6
commons-logging	log4j	5
js	rhino	5
xmlParserAPIs	xml-apis	5

TABLE II: The 15 most observed migrations from E

Computing the rule association graph leads to 34 connected components which means that we found 34 categories.

D. Valuable information from commits logs

As stated previously, we had to manually assign as either correct or wrong each migration rule found. Basically, one can search quickly over the internet the validity of the migration. But we consider that the developers are also willing to check the reasons why developers previously performed a migration. In a second time, they want to obtain additional information that support them to perform a migration.

To that purpose, we chose to mine commits logs of a wide set of Java open source projects. The idea is to look for commit logs that match both the source and the target of a given migration rule. Then, the results expose existing commit logs that explicit the migration and/or an associated justification. We arbitrarily selected software projects from four open-source platforms, 5340 projects from Google Code[14], 16000 from SourceForge[15], 12308 from Github[16] and 1063 from Apache[17]. We have collected the commits log of total of 34688 projects, resulting in around 3,9M of commit messages.

We provide on the web page[18] a search engine that returns a list of commits relevant for a given migration rule. To show that this tool can provide assistance for developers when they deal with migration rules, we explicit in Table III an excerpt of commits of interest for Logging and Database facilities. These migrations are discussed in Section IV.

To make sure about the invalidity of a migration rule, the absence of relevant results in the search engine is a solid argument. However, those messages must be used for indications only. There is no guarantee that a sound migration rule will match necessarily at least one meaningful commit log. The set of projects in the database has been chosen arbitrarily with no assumption about the quality of the registered commit logs. However, users are likely to find at least few results against a true migration rule.

[14]http://code.google.com/
[15]http://sourceforge.net/
[16]www.github.com
[17]http://svn.apache.org/repos/asf/
[18]http://www.labri.fr/perso/cteyton/search.php

Logging

- "logger api changed from `sfl4j-log4j` to `logback-classic`"
- "change logging system from using `log4j` directly to using `commons-logging`"
- "changed to use `commons-logging` instead of `log4j` directly"
- "moved ordi from `log4j` to `slf4j-api` to be more easily embeddable"
- "replaced `commons-logging` with `slf4j-log4j12`"
- "say goodbye `log4j`. welcome `slf4j-simple`"
- "loggers should be of type `org.slf4j` not `logback-classic`"

Database

- "replace `h2` w/`hsqldb` 2.0 for now"
- "changed database from `mysql` to `hsqldb` 2"
- "changed default database from `mysql` to `derby`"
- "use `derby` instead of `h2` database"
- "removed `derby` in favour of `hsqldb`"
- "move sql changes from `h2` to `mysql` engine"

TABLE III: Commit logs for assistance in rule validation (7 commits log for logging utilities and 6 for database facilities)

IV. Exploitation of the results

A. Analysis of the migration graphs

In this section we analyse the migration graphs extracted in Section III, and discuss the results we obtained regarding library migration. We compare the conclusion drawn from the graphs with the one that could be drawn from the plot of the evolution of the use of the libraries (this information is computed from the MCR). We show that our migration graphs offer an interesting complement of information and allow to detect interesting libraries which do not really stand out from the evolution plots. Then, we detail four visual patterns that help reading our migration graphs.

For sake of place, we present the analysis of five migration graphs attached to three domains : logging, database and XML. Those categories appear to be among the most utility libraries used in software projects. We have used *JUNG*[19] to draw them. As explained in Section III, we used $|g(m)|$ instead of $|r(m)|$ to label the arcs. Moreover, in the figures the size of the arcs is proportional to $|g(m)|$. Finally, the number of users of a library is indicated using the color of its node. A dark node indicates that the library is heavily used while a white node indicates that it has only few users. This number of users is relative to the library uses of a category, and is not related to the total number of projects we study.

1) Logging: Figure 5 shows a migration graph of libraries targeting the logging domain, containing 9 libraries. We can notice that a significant number of migrations go from *Commons-logging*[20] (80 departures) and *Log4J*[21](34). Inversely, the library *SLF4J*[22] appears to be a leader and is

[19]http://jung.sourceforge.net/
[20]http://commons.apache.org/logging/
[21]http://logging.apache.org/log4j/
[22]http://www.slf4j.org/

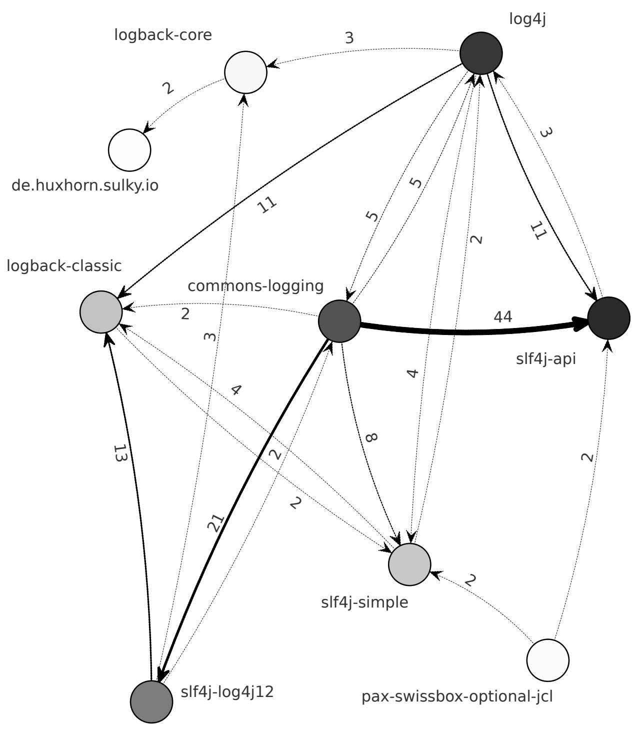


Fig. 5: Migration graphs for Logging libraries

the target of a high number of migrations. This is even more significant considering that *SLF4J* library is declined in several sub-libraries that appear in the graph with different nodes (e.g., *slf4j-simple*). Also, by computing the score of its respective ingoing edges, it turns out that 30 *groupIds* have migrated to *LOGBack*[23] and only 2 times from, targeting *SLF4J*. This may be an indication that projects that migrate to *LOGBack* are globally satisfied. Using this graph, we could recommend to the developers to consider dropping the *Commons-logging* and *Log4J* libraries, and to replace them by *SLF4J* or *LOGBack*. On the web pages of these two libraries we can read "*The logback-classic module can be assimilated to a significantly improved version of log4j*" and "*The Simple Logging Facade for Java or (SLF4J) serves as a simple facade or abstraction for various logging frameworks, e.g. java.util.logging, log4j and logback, allowing the end user to plug in the desired logging framework at deployment time*". It clearly seems that they aim to replace logging libraries such as *Commons-logging* and *Log4J*.

Figure 6 shows the evolution of the use of the four most-used logging libraries. In this figure, we can notice that *SLF4J* stands out from the other ones, which strengthens the conclusion drawn from the migration graph. However, *LOGBack* does not appear as particularly interesting and would probably not be seriously considered using only this plot.

2) Database: Within the database category, we observe several balanced migrations between *HSQLDB*[24], *Derby*[25], and

[23]http://logback.qos.ch/
[24]http://hsqldb.org/
[25]http://db.apache.org/derby/

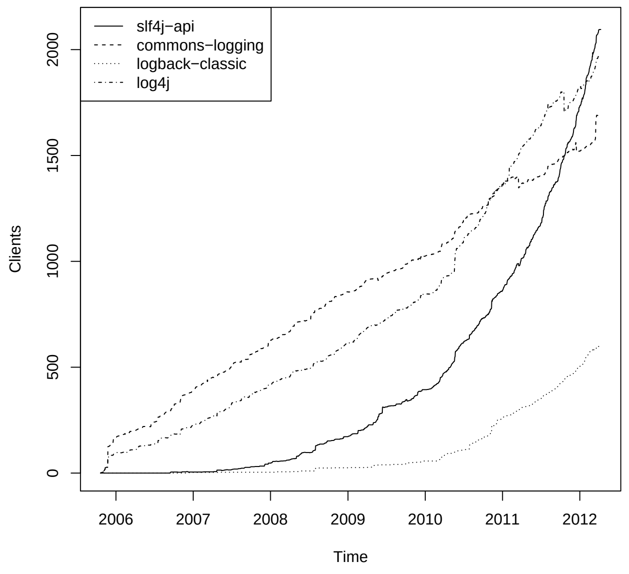


Fig. 6: Evolution of the use of the logging libraries

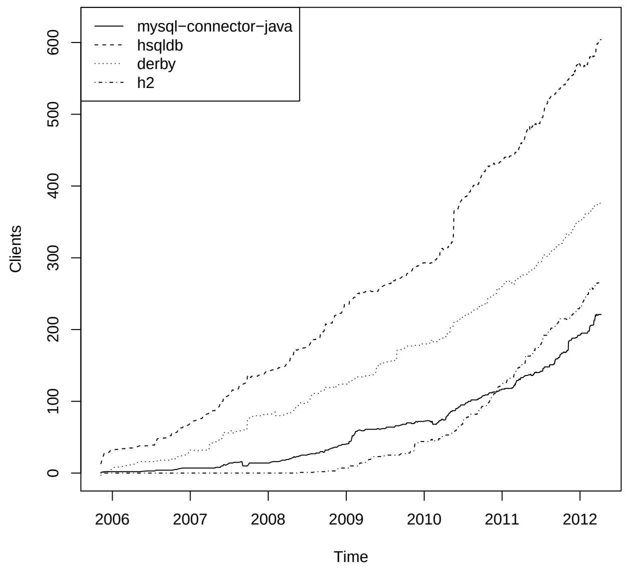


Fig. 8: Evolution of the use of the database libraries

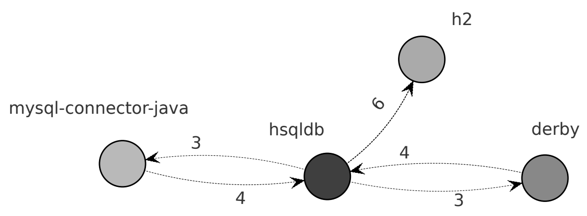


Fig. 7: Migration graph for the database libraries

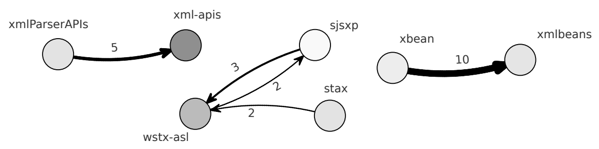


Fig. 9: Migration graph of the three XML categories

MySQL[26]. We also notice that 6 *groupIds* have migrated from *HSQLDB* to *H2*[27], and nobody did migrate from *H2*. By looking to this graph, we could recommend to the projects to seriously consider using *H2*. By looking on the web page of *H2*, it seems that it has more features[28] than its competitors, and very good performances[29]. Interestingly, we did perform a migration from *HSQLDB* to *H2* in one of our own software projects[30] when seeing these figures, with a significant gain in features and performances.

In contradiction to our approach, Figure 8 presents a classic evolution of the use of the database libraries. We can observe that *MySQL* interest is a bit slowing down, and that *HSQLDB* stands as the leader. Using only this plot, the *H2* library would not arise as interesting since it has less users than its competitors.

3) XML: Figure 9 shows the three migration graphs that can be attached to the XML domain. *XMLBeans*[31] looks like an emerging library as it underwent a migration from within 10 distinct *groupIds*. It seems to be the preferred target of the users of *XBeans*[32]. By looking to their websites, we did not understand the reason for this migration, although we have evidence in the SVN commit logs that it is not a false positive. *xml-apis* is preferred to *xmlParsersAPI*. By looking in the Maven repository, we noticed that in fact it is the same library, that was not eliminated by our text-mining techniques. Nevertheless, *xmlParsersAPI* is the outdated one. Finally, *SJSXP* and *Woodstox*[33] (wstx) are two STAX-compliant XML processors. As there are few users of *SJSXP*, and as some of them are migrating to *Woodstox*, the later seems to be the library of choice in for STAX compliant parsers. This intuition is strengthened when looking at Figure 10, which shows that *wstx* is the largest used STAX-compliant XML found with our migration rules.

B. Visual Patterns

To assist the analysis of a migration graph, we propose four visual patterns that can be quickly observed. These patterns allow to characterize some specific migrations. First, the *Gold Rush* pattern is observed when a library has many input migrations and few output ones. This clearly means that many projects migrate to this one. We believe this library

[26]http://www.mysql.com/
[27]http://www.h2database.com/html/main.html
[28]http://www.h2database.com/html/features.html
[29]http://www.h2database.com/html/performance.html
[30]http://harmony.googlecode.com
[31]http://xmlbeans.apache.org/
[32]http://www.xbeans.org/
[33]http://woodstox.codehaus.org/

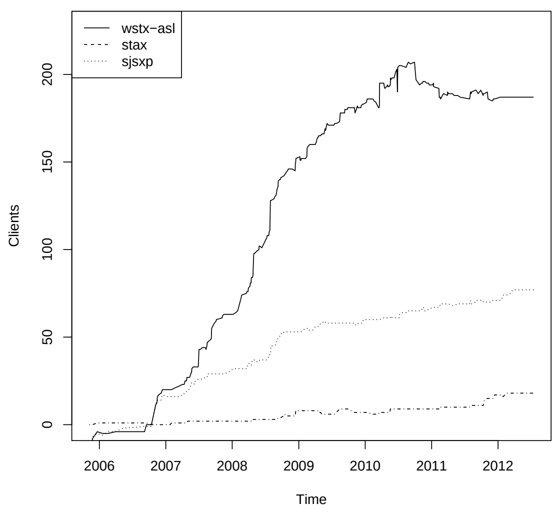


Fig. 10: Evolution of the use of stax-compliants libraries

has certainly more advantages than the other ones. As a consequence, the *Gold Rush* pattern can be used as a choice criteria to choose the right library to migrate to. On the contrary, we would not recommend projects that match the target of the *exodus* pattern, such as *commons-logging*. The *Pong* pattern refers to a bi-directional migration, that does not promote a project more than an other. The *Challenger* denotes a recent but promising candidate, as it has been adopted widely in a short period of time.

Table IV briefly summarizes these patterns. For each pattern, it explains how to observe the pattern, and provides an advice for developers that think about migrating their libraries.

V. Limits and Discussion

In this section, we discuss several important insights we gathered by performing our study.

Frequency of migrations: With regards to the data generated during the case study, we observe that library migrations happen along software life cycle. On the other hand, this phenomenon is not that frequent if we relate it to the total number of projects. It supports the idea that a large set of data is definitively necessary for a study similar to ours.

False positives: Even though we show confidence when we claim that two libraries can be employed for the same usage, the limited number of rules (80) classifies only a small part of the projects contained in the MCR that can be used as third-party libraries. However, more true rules are available in the initial set of 925 rules E, but except using a manual review, it seems hard to automatize a filter for them. Indeed, we observed many times in the low-ranked rules the rare rule syndrome: relevant rules appearing only once or twice. Recovering these rules would probably require additional information sources such as code source or documentation analysis. Also, a post-processing could be used to refine migration rules found using a software categorization technique, such as MudaBlue[1]. This could potentially eliminate a significant range of false positives.

Validation of the migration graphs: We did not perform an empirical validation of our migration graphs with developers. For the future, we plan to propose such graphs to developers and ask them to evaluate the usefulness of our recommendations.

Bi-directional patterns: A migration rule classified as *Pong* according to Table IV may reveal a situation where several projects migrate back and forth between two libraries. Currently, our approach does not correlate migration with time, but it could provide an useful additional indicator.

Impact of time: The mining of the migration rules is sensitive to the time. Indeed "recent" migrations necessarily have a lower score than the ones that took place over a long time period. Anyway it is not hard to overcome this issue by setting a threshold on the release date of the project release taken in consideration.

Incremental mining: It is possible to refresh the results of our mining approach periodically. Moreover it is possible to take advantage of the manual review of the rules to eliminate many false positives in the subsequent analyses. Indeed, if an expert rules exists among the candidates of a rule, it is possible to remove them from the candidates. Also, if a manually filtered rule is extracted, it is possible to not increment its score.

VI. Related work

Research work related to our concern falls in the domain of API migration and evolution.

Library categorization: Research has been done on software project categorization to allow searching for similar software. This problem is usually resolved by computing similarity score based on specific attributes, such as keyword identifiers as MudeBlue [1] does or API calls [2]. Those techniques require either the source code or the binaries versions of a set of libraries to compute similarity scores among them. Regarding our concern, the main limitation of such a work is that there is no guarantee than one library can be used instead of another one. For example, two software that perform XML operations could be seen as similar even if their intended usage is not.

Large-scale API usage: Mileva et al. have observed evolution of dependencies on 250 Apache projects along 2 years to mine usage of API versions [3]. The study shows the usage trends of different versions of a same project. It also was interested in cases where clients switched back to a previous version of a library. We reused the idea of usage trends in Section IV. While it offers valuable information, we have shown that several interesting libraries cannot be identified with just this piece of information.

Lämmel et al. propose a large-scale study on AST-based API-usage over a large set of open-source projects [4] . Their work provides an insight on how a specific API is globally used by client projects. In particular, they categorize whether

NAME	TARGET	DESCRIPTION	EXAMPLE
Gold Rush	Library	A library has many input migrations and few output ones. Developers should consider migrating to it.	SLF4J-API
Exodus	Library	A library has many output migrations and few input ones. Developers should consider to stop using it.	COMMONS-LOGGING
Challenger	Library	A library that is the target of a significant number of migrations originating from the most used library. Developers should consider migrating to it.	H2
Pong	Migration	Bi-directional migration rules with similar occurrences, no conclusion can be drawn from this pattern.	(HSQLDB,DERBY)

TABLE IV: Visual patterns observed in the migration graphs

the client calls the API (*library-like usage*) or if it inherits from it (*framework-like usage*). It may be interesting to integrate such an information in our library migration graph as some libraries may be better than other ones depending of client usage.

API Wrapping: During a library migration, the API-level challenge is to transform the code so that it becomes compliant with the new library. This domain aims at answering the question *"How to replace a library X with Y ?"*. Bartolomei et Al. have addressed this problem and studied the design of API Wrappers, which are objects that adapt and delegate the previous source code instructions towards the new API [5], [6]. The mappings are manually identified and their concern is to design such wrapper in order to obtain a compliant version of the new source code. Our approach is useful for such a problem as it identifies which libraries are source and target of migrations. It can then be used as a source of validation for the wrapper. Our approach does not study this particular aspect of research, but can help to identify projects that performed a particular migration.

API Upgrade: The problem of updating a library has also been studied in the literature. A challenge w.r.t. to library usage is to provide relevant snippets of code source according to the programmer's context. We distinguish two main techniques to that extent. The first one mines code that already performed an update. For instance, Schafer et Al. [7] examined code instantiations of two versions of a framework. This code is included with the release as test or example code. Also, SemDiff [8] is a client-server connected to a framework source code repository that mines the changes and recommends modifications for a client migration. The second variety of approach requires only internal code of two API versions and applies origin analysis techniques. Hence, a graph-based representation of the code based on dependencies allows for element matchings from the two versions. Some promising results have been achieved in this area [9][10]. Whatever the technique, our approach can be used as a mass source of data to get real library migrations and to get references of real projects that do have performed migrations. Such a mass of data can be used to validate the proposed approaches.

Language migration: Zhong et al. proposed a Mining API Mapping approach that detects relations from two versions of an API written in different languages[11]. The idea is to get client-code from the two versions and to build a transformation graph that represents the API-usage migration from one language to another. Zheng et al. propose a cross-library recommendation tool based on Web queries [12]. The idea is to inquire Web search engines and to mine results proposed from the query. One example of query could be "HashMap C#" when looking for the equivalent for standard Java HashMap for C#. The results are computed one by one and candidates are ranked by relevance, mainly according to their frequency of appearance. For the moment this work provides only preliminary results and queries proposed are of a coarse grain. Also, it strongly lies on Web search engines such as Google, and requires manual query writing, which can highly influence the results. Regarding our approach, this work can be used to merge equivalent libraries and then to improve library migration graphs.

VII. CONCLUSION AND FUTURE WORK

In this paper we address the issues raised by library migration, which is a mandatory task during the life cycle of any project that depends on open source libraries. When a project has to replace one of the libraries it depends on, two issues have to be addressed. First, all libraries that provide similar facilities have to be identified. Second, the substitute candidate that fits the project's needs has to be chosen among the identified libraries.

To face these two issues we propose a mining process that is based on library dependencies and their evolutions. We provide what we call migration graphs that show categories of similar libraries and that exhibit how projects migrate between them. Thanks to these graphs, one can quickly identifies which libraries are candidates to migrate to and how many projects have chosen them to migrate from.

As a validation, we have applied our process on the Maven Central Repository that tracks the dependencies of a large number of projects. We have generated migration graphs for the different categories of libraries. For sake of place, we have presented only graphs that correspond to the logging, database and XML categories. Those graphs present libraries of these technical domains and clearly show which are the preferred libraries to migrate to. A deep analysis of these graphs show the insights they provide for library migration in comparison to classical measures such as usage trends.

The work presented in this paper uncovered the fact that library migration is not a frequent phenomenon through open source software life cycle. In our opinion, this is mainly due to two points. First, the cost of performing the migration by updating the project source code base. Second, the lack of existing build automation tools to propose migration recommendations. This raises the developers level of uncertainty to assess the benefits of choosing a new library for their project.

As a further work, we first plan to apply our process on other sources of data. In particular, we can extend our study to the entire open source software world, similar to large-scale repositories mining approaches proposed with SourcererDB [13] or SeCOLD [14]. Moreover, tuning our approach the commit granularity level of version control systems would allow to target more precisely migration rules.

We plan to apply our study to other dependencies management systems, to show that the proposed approach can be generalized. For instance, Ubuntu packages[34] offer a convenient way to access their list of dependencies. As packages evolve across Ubuntu versions, we should reproduce our experiment on such system. Such a study should lead to package migration graphs that would show migrations of packages along Ubuntu releases.

Finally, we plan to use our approach to assist developers while they migrate their code to become compliant with a new library. As our approach identifies existing projects that already did the migration task, we plan to analyse the source code of these projects before and after the migration in order to detect migration patterns. Such patterns abstract refactoring actions that must be performed to be compliant with the new library. The goal is then to automatically apply them in new projects that are migrating.

References

[1] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "Mudablue: an automatic categorization system for open source repositories," *J. Syst. Softw.*, vol. 79, no. 7, pp. 939–953, Jul. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2005.06.044

[2] C. McMillan, M. Linares-Vasquez, D. Poshyvanyk, and M. Grechanik, "Categorizing software applications for maintenance," in *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ser. ICSM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 343–352. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2011.6080801

[3] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining trends of library usage," in *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, ser. IWPSE-Evol '09. New York, NY, USA: ACM, 2009, pp. 57–62.

[4] R. Lämmel, E. Pek, and J. Starek, "Large-scale, ast-based api-usage analysis of open-source java projects," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, ser. SAC '11. New York, NY, USA: ACM, 2011, pp. 1317–1324. [Online]. Available: http://doi.acm.org/10.1145/1982185.1982471

[5] T. Tonelli Bartolomei, K. Czarnecki, R. Lämmel, and T. van der Storm, "Study of an api migration for two xml apis," in *2nd International Conference on Software Language Engineering (SLE)*, vol. 5969/2010, Denver, USA, 10/2009 2009, pp. 42–61.

[6] T. Tonelli Bartolomei, K. Czarnecki, and R. Lämmel, "Swing to swt and back: Patterns for api migration by wrapping," in *26th IEEE International Conference on Software Maintenance (ICSM)*, Timisoara, Romania, 09/2010 2010.

[7] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 471–480. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368153

[8] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 481–490. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368154

[9] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 325–334. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806848

[10] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," *SIGPLAN Not.*, vol. 45, pp. 302–321, October 2010. [Online]. Available: http://doi.acm.org/10.1145/1932682.1869486

[11] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining api mapping for language migration," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 195–204. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806831

[12] W. Zheng, Q. Zhang, and M. Lyu, "Cross-library api recommendation using web search engines," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 480–483. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025197

[13] J. Ossher, S. K. Bajracharya, E. Linstead, P. Baldi, and C. V. Lopes, "Sourcererdb: An aggregated repository of statically analyzed and cross-linked open source java projects," in *MSR*, M. W. Godfrey and J. Whitehead, Eds. IEEE, 2009, pp. 183–186.

[14] I. Keivanloo, C. Forbes, A. Hmood, M. Erfani, C. Neal, G. Peristerakis, and J. Rilling, "A linked data platform for mining software repositories," in *MSR*, M. Lanza, M. D. Penta, and T. Xi, Eds. IEEE, 2012, pp. 32–35.

[34]http://packages.ubuntu.com/