Automatic Discovery of Function Mappings between Similar Libraries

Cédric Teyton, Jean-Rémy Falleri, Xavier Blanc
Univ. Bordeaux, LaBRI, UMR 5800
F-33400 Talence, France
{cteyton,falleri,xblanc}@labri.fr

***Abstract*—*Library migration* is the process of replacing a third-party library in favor of a competing one during software maintenance. The process of transforming a software source code to become compliant with a new library is cumbersome and error-prone. Indeed, developers have to understand a new Application Programming Interface (API) and search for the right replacements for the functions they use from the old library. As the two libraries are independent, the functions may have totally different structures and names, making the search of mappings very difficult. To assist the developers in this difficult task, we introduce an approach that analyzes source code changes from software projects that already underwent a given library migration to extract *mappings* between functions. We demonstrate the applicability of our approach on several library migrations performed on the Java open source software projects.**

I. Introduction

Software systems depend more and more on third party libraries that provide robust and efficient functionalities [1]. Using libraries saves development time as it prevents developers to redevelop existing features. However, as both software systems and libraries irremediably evolve in their own direction, developers sometime need to replace some used libraries by another ones, for maintenance reasons. This phenomenon is known as *library migration* and has been studied and observed on Open Source Software (OSS) projects [2], [17].

When a developer has to perform a migration between two libraries L_A and L_B, she has to translate all the dependencies of L_A into dependencies of L_B. In other words, if we consider that a library provides a set of functions, she has to find for each function she use in L_A the corresponding function(s) in L_B. Such a task is known to be tedious and error prone, in particular if she does not know any of the function of L_B [3]. The main challenge is then to identify the mappings between the functions provided by the libraries.

When a migration is done between two versions of a same library, this challenge can be faced by a syntactic analysis of the two versions as they often only slightly differ on their syntax and/or structure [4], [5], [6], [13], [8]. When a migration is done between two different libraries, then the challenge cannot be faced with the same techniques as the structure and syntax of the libraries are significantly different.

In this paper, we propose to address the problem of the identification of mappings between two different libraries. We assume that libraries L_A and L_B respectively provide N and M functions. Our goal is then to identify for each function of L_A the set of corresponding functions of L_B, if it exists. As no hypothesis can be established regarding the existence of mappings between L_A and L_B, there may be functions of L_A that have no corresponding function in L_B. Moreover, some functions of L_A may have several corresponding functions in L_B. In particular, a function f in L_A may have two corresponding functions g and h in L_B but with different constraints (g AND h, g OR h).

Our proposal consists in extracting function mappings by mining existing software projects that have already performed library migrations. The main idea is to identify commits where the migrations have been done and then to analyze the changes that have been performed to extract function mappings. For instance, Listing 1 presents a commit where a migration has been performed between the libraries *commons.lang* and *guava*. An analysis of the commit can infer a mapping between the function *Validate.notNull(int)* and *Preconditions.checkArgument(boolean)*.

Listing 1: An example of migration commons.lang → guava.lang.

```
-import org.apache.commons.lang.Validate;
+import com.google.common.base.Preconditions;

public long getProblemVersion(String id) {
-    Validate.notNull(id);
+    Preconditions.checkArgument(id != null);
}
```

Several issues have to be faced to achieve our proposal. First, software projects that replaced their libraries have to be identified as well as their commits where migrations have been performed. Second, an analysis of the commits have to be performed to detect library migrations. Third, a process of knowledge extraction has to be deployed to reveal the existence of mappings.

The contribution of this paper is then threefold. First, we provide an efficient approach to identify any library migration through a project history. The library migration is localized in a period of time referred as a *migration segment*. Secondly, within such segment we use textual differencing between edited source code files to identify with precision changed parts of code. We search for *hunks* that contain both removed and added function invocations from the source and target libraries of the migration. The functions extracted in these hunks form candidate *function mappings*. The mappings are then manually

reviewed to evaluate the efficiency of our approach. Finally, we propose a lightweight filtering method to improve the precision of the approach.

We applied our approach on a huge corpus of Java OSS projects, looking for migrations between five pairs of commonly used Java libraries. The quality and the precision of the results are discussed in this paper, and show that our approach is promising and produces valuable information.

The remainder of this paper is structured as follows. Section II presents the related work. Section III describes the abstract model of our approach and its general process. Section IV describes how we applied our approach for the Java programming language and OSS projects. Section V discusses the results obtained from the experiments. Finally Section VI seals the paper.

II. Related Work

To the best of our knowledge, no existing approach tackles the problem of thoroughly finding mappings between the functions of two independent libraries (the *library migration* problem), however there are many approaches that find mappings between the functions from two different versions of a same library (the *library update* problem). Therefore we divide the related work in three parts. In the first one we describe the library update approaches while in the second one we describe the few approaches that considered the library migration problem. In a last part, we introduce the problem of discovering similar libraries.

A. Library Update

A large number of approaches use only the source code of the concerned library to deduce mappings. They use several techniques to extract the mappings. Textual similarity of methods signatures is commonly used [5], [9], [10], [11]. Computing the similarity of the sets of callers of the functions is also a very common technique [5], [8], [11]. Computing the similarity of the source code of the functions has also been tried [12], [11]. Finally inferring refactorings from changes of the library structure has also been investigated [13], [9]. Several approaches use a combination of these techniques [5], [11], [9].

All these techniques are pointless in our context. Indeed, since the libraries are independent, their names, source codes, and structures are likely to be significantly different. Also, comparing the similarity of the callers of the functions of the libraries is not possible.

Cossette et al. [14] performed a retroactive study on several library updates manually identified. They recorded the different adaptations they had to make. They show that there is no silver bullet because each of the different techniques described above fails in some situations. They also show that for a significant number of situations, none of the above technique work.

Two approaches of this line of work paved the way of our approach, because they use client code of the libraries to deduce mappings. Nguyen et al. [9] relies first on mappings extracted using textual and structure similarities of the functions and then on usage graphs extracted from the client. Unfortunately, this approach is not applicable in our context because it strongly relies on the first step which uses structure and textual similarity. Schäfer et al. [6] analyses changes of the call sites in the client projects of the library to extract transactions of removed functions and added functions. Then, they compute association rules from these transactions. Finally they filter the produced rules by using the textual similarity of the function signatures. The first step of this approach has largely inspired us, but in their work they consider a whole method as a single call site, which can generate a lot of false positives for large methods. It can also miss a lot of mappings in case of renaming of the enclosing methods. We improved this part of the process by computing the call sites using textual differencing techniques. Also, the filtering process they apply is not adapted to our context where the names are very different, therefore we introduced a filtering process that does not use this information.

The library update problem has also been studied in a context where the system used differs. Winter et al. shown that Java Libraries that run on standard Java Virtual Machine (JVM) may be incompatible on JVM running on embedded systems [**?**]. They propose program transformation to eliminate parts of the API that are incompatible on such systems, like floating points operations or multi-threading. This approach is nonetheless not enough related to our context to provide elements of comparison.

Finally, in a similar context Kapur et al. propose Trident, a IDE plugin to assist references refactoring in a library update context [**?**]. Their work addresses the process of transforming a project source code assuming that the list of changes between two library versions are known. This tool could be investigated to evaluate its convenience with two independent API. The main impediment is to preliminary compute all the equivalence between such two libraries, which is highly complex as shown in this paper.

B. Library Migration

Two existing techniques have considered the problem of library migration. First, Zheng et al. proposed a cross-library recommendation tool based on Web queries using search engines such as Google [**?**]. The idea is to leverage the large database of knowledge over the Web to search for mappings between two API. An instance of query could be *"HashMap C#"* if looking for the equivalent for standard Java HashMap for C#. In our context, this approach can hardly be automatized since queries should be constructed from method signatures. It is thus unlikely to get satisfying results with such queries. Moreover, deriving a human readable query from a method signature, e.g. *"Check string is not empty Guava"*, is not straightforward. Therefore, it is orthogonal to our approach.

Secondly, Bartolomei et al. [15], [16] introduced an approach that consists in never calling directly a library but instead using a "wrapper" that calls the functions of the library. Then, migrating to another library requires to write a new wrapper with the same interface as the previous one but that calls the functions of the new library instead. The main drawback of this

approach is that it requires to have anticipated the migration of the library from the beginning, which is not often the case. On the other hand, it can be used to safely migrate from one library to the other when the need for the migration has been identified, by constructing a wrapper before the migration. Then writing the wrapper for the new library can be easier with our approach.

In a previous work, we studied the general phenomenon of library migration on open source projects [2], [17]. We performed a large scale mining of the evolution of the projects of the Maven repository to identify common library migrations. We leveraged on this work to gather a corpus of projects that performed library migrations, and to set the thresholds used in our approach.

C. Similar Libraries

The general approach presented in this paper takes as input two similar but independent libraries. However, selecting a satisfying library to migrate to is not a trivial task. In this context, discovering existing similar libraries can be a requirement for a developer that consider library migration. In addition to our previous work [2], we mention CLAN, an approach that categorizes a set of software systems based on their internal content and API calls [**?**]. Similar software are likely to be used in a same context, and this provide libraries to migrate to.

III. Approach

This section describes our approach to identify function mappings between two similar libraries by observing software projects that already underwent the migration. We start by describing an abstract model of projects having multiple versions and using several libraries. Then we explain how we efficiently search in the versions of a project to discover which library migrations it underwent. We then describe the fine-grained analysis we perform on the source code to extract the function mappings. Finally, we describe a filtering technique to improve the precision of the extracted mappings.

A. Preliminary Definitions

We abstract the data needed to perform our analysis in a simple model, a set of software projects and their list of versions, and for each version the associated set of library dependencies.

Definition 1 (Library): Let L be the set of libraries. Each library $l \in L$ provides a set of exported functions F_l. For any libraries $l, l' \in L$ with $l \neq l'$, we have $F_l \cap F_{l'} = \emptyset$.

For instance, assume two logging libraries $\mathrm{log4j}$ and $\mathrm{slf4j}$ that both provide 3 functions, shown in Table I.

Definition 2 (Project, versions and dependencies): Let P be the set of analyzed software projects. For each project $p \in P$ there is an associated totally ordered set of versions $V_p \subset \mathbb{N}$. Versions are sorted chronologically according to their date. In our context, the versions of a project are the commits from the version control system. For a project $p \in P$ at version $i \in V_p$, we define its library dependencies $\mathrm{dep}_p(i) : V_p \to \mathcal{P}(L)$.

Table II illustrates this model with 3 versions of a project `Foo` and their associated libraries. We have thus $\mathrm{dep}_p(0) = \{\mathrm{junit}, \mathrm{log4j}\}$ or $\mathrm{dep}_p(2) = \{\mathrm{log4j}, \mathrm{slf4j}, \mathrm{junit}\}$.

The next part of our model defines migration rules and migrations.

Definition 3 (Migration rule): Let M be the set of valid migration rules. $M \subseteq L^2$ is a set of pairs (s, t). Such a pair indicates that the library t can replace the library s. Note that this relation is symmetric, i.e. $(t, s) \in M$ iff $(s, t) \in M$. Therefore, we will denote such a rule $s \leftrightarrow t$. Finally $\mathrm{lib}(M)$ denote all the libraries that can be replaced (i.e. contained in a rule of M).

Definition 4 (Migration): A migration $s \to t$ with $s, t \in L$, is performed by a software project p between two versions $i, j \in V_p$ with $i < j$ when the following condition holds:

$$s \leftrightarrow t \in M \wedge s \in \mathrm{dep}_p(i) \wedge t \notin \mathrm{dep}_p(i) \wedge s \notin \mathrm{dep}_p(j) \wedge t \in \mathrm{dep}_p(j) \quad (1)$$

In other words, p uses a subset of F_s at version v_i. At version v_j, p uses functions from F_t but does not use functions from F_s any longer. Additionally, we require that (s, t) is a valid library migration. In our example, this situation holds since $\mathrm{log4j} \in \mathrm{dep}_p(1)$, $\mathrm{slf4j} \notin \mathrm{dep}_p(1)$, $\mathrm{log4j} \notin \mathrm{dep}_p(3)$ and $\mathrm{slf4j} \in \mathrm{dep}_p(3)$. Since $\mathrm{slf4j} \leftrightarrow \mathrm{log4j}$ is a valid migration rule, we have therefore a migration $\mathrm{log4j} \to \mathrm{slf4j}$ between version the versions 1 and 3.

Definition 5 (Migration segment): A migration segment is a pair of versions (i, j) with $i, j \in V_p$ and $i < j$ where a migration $s \to t$ has been observed. A migration segment is the shortest interval where a given migration can be observed.

For instance, in our example, $(1, 3)$ is a migration segment for the migration $\mathrm{log4j} \to \mathrm{slf4j}$, while $(1, 4)$ is not whereas it contains the migration.

B. Extracting Function Mappings

To extract mappings between functions of two libraries, we use a two-step process. First we apply an efficient search algorithm to find the migration segments. Then, within a migration segment, we apply a fine-grained code analysis, based on textual differencing, to extract the mappings between the functions of the libraries.

1) Extracting Migration Segments: The first step of our approach consists in identifying migration segments from a set of versions V_p of a project p. When searching for migration segments, the longest operation is to extract the

TABLE I: Two libraries that provide 3 public functions.

Library	Symbols
`log4j`	Logger.debug(String) Logger.error(String) Logger.getLogger(Class)
`slf4j`	Log.debug(Object) Log.fatal(Object) Log.getLog(Class)

TABLE II: A sample project `Foo` with its versions and dependencies.

Version	Dependencies
0	{junit}
1	{log4j, junit}
2	{log4j, slf4j, junit}
3	{slf4j, junit}
4	{slf4j, junit}

set of libraries $\mathrm{dep}_p(i)$ of the project's versions, because it requires the downloading and the analysis of the source code of the whole version. Extracting all migrations segments contained in a set of versions requires the analysis of the dependencies of each version, which is very expensive in term of computation time, as shown in Section IV. To reduce this computation time, we introduce an approximate divide and conquer algorithm.

Our algorithm is based on several hypotheses. The first one is that libraries can be replaced but never dropped. Therefore if there is a migration contained in an interval of versions, the target library has to be a dependency of the last version of the interval. We also assume that in a project, there are many intervals of versions where the dependencies do not change, as it is not very common to introduce or replace a library. Another hypothesis is that it is very unlikely to have more than one migration in a short time span (i.e $l_a \rightarrow l_b \rightarrow l_c$) and that rollback of libraries (i.e $l_a \rightarrow l_b \rightarrow l_a$) almost never happens. . Finally, the last hypothesis is that there are few migration segments in a project history, with short lengths.

Algorithm 1 shows our divide and conquer algorithm to extract migration segments based upon our hypotheses. It takes as input two versions i and j with $i < j$ and a set of segments. It works as follows: if $\mathrm{dep}_p(i) = \mathrm{dep}_p(j)$ or $\mathrm{dep}_p(j) \cap \mathrm{lib}(M)$ is empty, the search ends between i and j. Else, if $j - i \leq t_{min}$ we check if a valid rule can be inferred from the dependencies of i and j. If it is the case, we use a binary search algorithm to find the exact bounds of the segment and we add it into the set of segments. If $j - i > t_{min}$, the algorithm is recursively applied on the intervals $(i, \frac{i+j}{2})$ and $(\frac{i+j}{2}, j)$. The drawback of our algorithm is that it can miss migration segments when there are rollbacks, or when the splitting point of the interval v_m is located within a migration segment. However we have shown that in practice, there are few segments in a set of versions, and they have a short length [17]. Therefore the probability of these situations is very low.

To better understand this algorithm, assume the example of Figure 1 with a project p having 7 versions. We use a distance threshold $t_{\min}$ of 3.The libraries associated to version v_0 and v_6 are different and there are two libraries at v_6 that are possible migration targets, so we divide the interval in two and apply the search on (v_0, v_3) and (v_3, v_6). Since $\mathrm{dep}_p(v_3) = \mathrm{dep}_p(v_6)$, the recursion ends. Since $\mathrm{dep}_p(v_0) \neq \mathrm{dep}_p(v_3)$ and $3 - 0 \leq t_{\min}$ we compute $\mathrm{rem} = \{\mathrm{log4j}\}$ and $\mathrm{add} = \{\mathrm{slf4j}\}$. We have $\mathrm{log4j} \leftrightarrow \mathrm{slf4j} \in M$, therefore `extract` is called with the parameters v_0, v_3, log4j and slf4j. Finally the segment $(2, 3)$ is extracted. In this example, 4 versions have been analyzed

Algorithm 1 Migration segment extraction

Require: p : a project
Require: V_p : the set of versions of p
Require: t_{min} : a minimal length
Require: M : the set of valid library migration rules

```
Seg ← ∅
FIND_SEGMENT(v_0, v_head, Seg)
return Seg
function FIND_SEGMENT(v_i, v_j, Seg)
    if dep_p(v_j) ∩ lib(M) = ∅ ∨ dep_p(v_i) = dep_p(v_j) then
        return
    else if (j − i) ≤ t_min then
        rem = dep_p(v_i) \ dep_p(v_j)
        add = dep_p(v_j) \ dep_p(v_i)
        for all s ∈ rem do
            for all t ∈ add do
                if s ↔ t ∈ M then
                    Seg ← Seg ∪ EXTRACT(v_i, v_j, s, t)
                    return
                end if
            end for
        end for
    else
        v_m ← (v_i + v_j) / 2
        FIND_SEGMENT(v_i, v_m, Seg)
        FIND_SEGMENT(v_m, v_j, Seg)
    end if
end function
```

(including the one analyzed by the `extract` procedure). The exact approach to detect migrations would need to analyze the 7 versions. The performances of our approximate algorithm compared to the exact algorithm are discussed in Section IV.

2) Extracting Function Mappings: A migration segment is delimited by a couple (i, j) of project versions and targets a migration $s \rightarrow t$. To extract function mappings, we use a fine-grained analysis of the source code changes between each pairs of successive versions between i and j. The technique proposed below is based on the assumptions that during a library migration task, developers are likely to replace the calls of functions of F_s by calls functions from F_t.

Consequently, we compute the textual differences between two versions of an edited source code file. We assume that for every pair of versions $(k, k+1)$ with $i \leq k \leq (j-1)$, the list of edited source code files of the version $k+1$ can be retrieved. We use the standard Unix *diff* tool [18] to compare two source files and extract a list of *hunks* that summarize the changes that produce the file at version $k+1$ from the file at version k. A hunk is a sequence of either removed, added, or removed and added lines of code. It contains a header with the line positions of both the removed lines in the *old* version of the file and added lines in the *new* file. Note that either a line number or a range of two lines can be specified in the hunk header. Listing 2c shows the hunks computed from the source

code of Listing 2a and Listing 2b. We only retain the hunks containing both removed and added lines, and we record the position of the removed (resp. added) lines in the old (resp. new) version of the file.

Then we parse the two versions of the source code file at v_k and v_{k+1} to collect the line positions of functions calls to F_s and F_t respectively. We retain only hunks h for which 1) the removed lines contain at least one call to a function of F_s 2) the added lines contain at least one call to a function of F_t. We denote $\mathrm{rem}(h)$ (resp. $\mathrm{add}(h)$) the list of functions of F_s that were removed (resp. added) in h. For such a hunk, we extract as the function mappings the Cartesian product $\mathrm{rem}(h) \times \mathrm{add}(h)$.

Definition 6 (Function mapping): Given a migration rule $s \leftrightarrow t \in M$, a function mapping $x \leftrightarrow y$, with $x \in F_s$ and $y \in F_t$, indicates that a similarity exists between x and y. Thus, y is a possible replacement for x. Similarly to the migration rules, this relation is symmetric. The score $sc(x, y)$ of a mapping rule is the number of hunks where the rule has been observed.

To illustrate, let us assume two versions 1 and 2 of the file *Bar.java* shown in Listing 2a and Listing 2b. The corresponding *diff* is composed of 3 hunks and is shown in Listing 2c (they are indicated by the @@ symbols). The three hunks are retained because they contain removed calls to functions from F_s and added calls to functions from F_t. They generate the mappings $\mathrm{Log.getLog(String)} \leftrightarrow \mathrm{Logger.getLogger(String)}$ (first and second hunks) and $\mathrm{Log.fatal(String)} \leftrightarrow \mathrm{Log.error(String)}$ (third hunk). This example shows the benefit of using hunks to extract function mappings. For the last hunks, the function `something(int)` has been renamed, but we are still able to extract mappings. Also, it prevented to extract mappings $\mathrm{Log.getLog(String)} \leftrightarrow \mathrm{Log.error(String)}$ in the last method because it was separated into two different hunks (the second and the third).

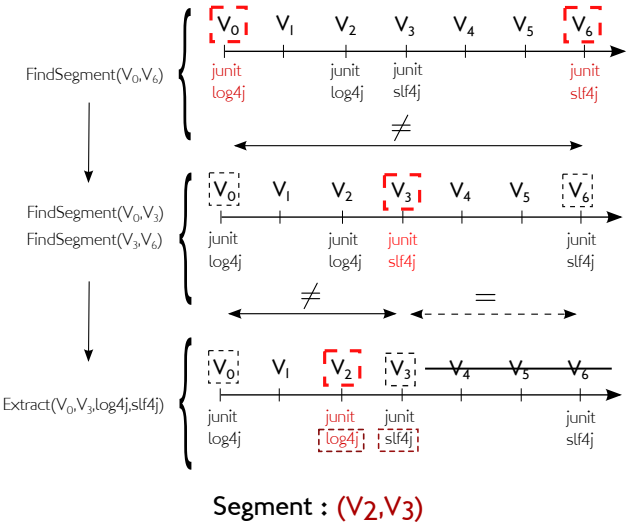

Fig. 1: Illustration of Algorithm 1 on the versions of our sample project of Table II. Here, $t_{min} = 3$.

C. Function Mappings Filtering

Computing the Cartesian product has the drawback to generate false positives. In addition, textual differencing cannot guarantee an optimal precision, since the modifications may involve many consecutive lines. We thus propose a lightweight filtering technique that aims to reduce the number of false positives. We use the technique proposed by Melnik et al. [19] based on relative similarities, because it can cope with n:m mappings. The idea is to identify elements that have a strong mutual similarity when considering the whole population. We assume a set of function mappings. Each mapping $s \leftrightarrow t$ is assigned the $sc(s, t)$ similarity score, also referred as their absolute similarity.

Next, the set of rules is transformed into an undirected graph where each node is a function. There is an edge between two nodes s and t if a candidate mapping rule $s \leftrightarrow t$ exists. Then, for each node n in the graph, we record the best absolute similarity score max_n it has with its connected nodes. A new graph is constructed by labeling an edge (s, t) with the two relatives scores for $\frac{sc(s,t)}{\max_s}$ and $\frac{sc(s,t)}{\max_t}$, denoted respectively $\alpha(s, t)$ and $\alpha(t, s)$. The final graph is composed of relative similarity scores that are values between 0 and 1.

To better clarify how this process operates, let us consider the example shown in Figure 3. There are 4 candidate mapping rules associated with their absolute similarities. The top right graph represents the relative similarities computed by fractions for each node and its respective best absolute similarity. The resulting new values are displayed in the bottom graph. At this step, each mapping rule is associated to two relative similarity scores.

To finish, we process the candidate mappings to select pairs (s, t) of elements having a relative similarity of at least a threshold value $t_{\mathrm{rel}} \in [0, 1]$. This means that both $\alpha(s, t)$ and $\alpha(t, s)$ have to be greater or equal to t_{rel} to validate the candidate mapping $s \leftrightarrow t$. The threshold has to be set up between 0 and 1.

In our example, setting a threshold t_{rel} to 0.5 will retain only the mappings $\mathrm{log4j.a()} \leftrightarrow \mathrm{slf4j.x()}$ and $\mathrm{log4j.b()} \leftrightarrow \mathrm{slf4j.y()}$. Inversely, fixing a value of 0.25 will validate all the candidate mappings, while only the mapping $\mathrm{log4j.a()} \leftrightarrow \mathrm{slf4j.y()}$ is filtered if t_{rel} is set to 0.3. The impact of the choice of t_{rel} is evaluated in Section IV.

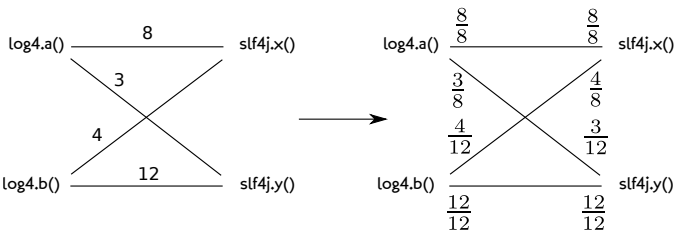

Fig. 3: Computing relative similarities for candidate mappings.

```java
public class Bar {

 public void test() {
  Log.getLog("MyLogger");
  something(3);
 }

 public void something(int i) {
  Log.getLog("MyLogger");
  if (i > 0) {
   Log.fatal("Error");
  }
 }
}
```

(a) Bar.java - version 1

```java
public class Bar {

 public void test() {
  Logger.getLogger("MyLogger");
  other(3);
 }

 public void other(int i) {
  Logger.getLogger("MyLogger");
  if (i > 0) {
   Logger.error("Error");
  }
 }
}
```

(b) Bar.java - version 2

```
@@ -4,2 +4,2 @@
-  Log.getLog("MyLogger");
-  something(3);
+  Logger.getLogger("MyLogger");
+  other(3);
@@ -8,2 +8,2 @@
- public void something(int i) {
-  Log.getLog("MyLogger");
+ public void other(int i) {
+  Logger.getLogger("MyLogger");
@@ -11 +11 @@
-   Log.fatal("Error");
+   Logger.error("Error");
```

(c) Diff Bar.java 1-2

Fig. 2: A sample class `Bar` that migrates from slf4j to log4j.

IV. Empirical Evaluation

In this section we apply our approach on several open source software projects written in Java to answer the following research questions:

- Does our segment extraction algorithm find similar results in comparison with an *exact* algorithm and is it faster? (RQ1)
- Does our function mapping extraction technique using hunks as call sites give better results than the one using methods as call sites? (RQ2)
- What is the name similarity and the amount of multi-mappings within the function mappings? (RQ3)
- Does our filtering technique improve the precision of the results? (RQ4)

We first present our corpus of projects and library migration rules. Then we describe our tool implementation. Finally we present how we extracted a set of manually checked function mappings.

A. Setup

1) Library Migration Rules and Projects: We choose the Java programming language for this study for two reasons. First, library usage is common in Java software development. Second, library usage detection is a trivial process using a static source code analysis. For this survey we selected four popular migration rules, as shown in our previous study [2]. This subset covers different domains of usage and their popularity increases the likeliness to observe data for our study. The first two ones are included in the well-known projects Apache Commons and Google Guava. Apache Commons "*is an Apache project focused on all aspects of reusable Java components*", while Google Guava "*contains several of Google's core libraries that we rely on in Java-based projects*". In other words, Commons and Guava extend or re-implement functionalities provided by the Java standard library. We focus on two migration rules within these projects: guava.io ↔ commons.io (called *I/O*) and guava.lang ↔ commons.lang (called *Lang*). The third migration rule is between two libraries that manipulate JSON documents: the standard *org.json* library, and the Google *gson* library. This rule, org.json ↔ gson is called *JSON*. Finally, the last migration rule is called *Mock* and is between two testing libraries that support the writing of tests containing *mock* objects: *jmock* and *mockito* (jmock ↔ mockito).

To perform our study, we also need a large set of projects that already performed the migrations described above. To retrieve such a set, we queried the Github, GoogleCode and Sourceforge open-source project hosting platforms for Java projects. We then randomly selected 14,000 projects from which discarded 2,402 empty projects, leading to a corpus of 11,598 Java projects.

2) Instrumentation: We used the framework Harmony [**?**] to extract the versions of the projects from our corpus. It supports Git, SVN and Mercurial version control systems and constructs the history of any project in a suitable way for any type of analysis on the history.

We used the tool ScanLib[1] to discover the set of third-party libraries used by a project. ScanLib extracts an Abstract Syntax Tree (AST) for each Java file of a project (using Eclipse JDT), and traverses it to look for qualified names in the source code. Whenever it finds a qualified name, it checks in its internal database if the qualified name matches a regular expression of a known library. For instance, the library *mockito* is attached to the regular expression `org.mockito.*`. The internal database of ScanLib has been carefully designed in order to have at most one library for a given qualified name. Additional details on the construction on the database construction are discussed in [17].

We use the standard Unix diff tool to compute the hunks between the two versions of a modified file. We record the lines corresponding to the hunks, and use Eclipse JDT to extract the function calls contained in these lines with an AST traversal.

B. Function Mappings

We now detail how we constituted a set of function mappings by executing our approach on the corpus of libraries and projects.

1) Extracting Migration Segments: To identify migration segments, a minimum distance value of $t_{\min} = 25$ for the migration segments has been set up for all the experiments

[1]https://code.google.com/p/scanlib-java/

TABLE III: Number of segments, hunks and functions extracted from the corpus of projects.

Rule	#Segments	#Hunks	#Functions
I/O	12	33	31
Lang	14	66	55
JSON	4	99	48
Mock	6	87	35

TABLE IV: Number of functions in the hunks.

	Number of hunks containing n functions			
Rule	0-2 Func.	3 Func.	4-6 Func.	7-10 Func.
I/O	31	2	0	0
Lang	39	8	19	0
JSON	51	7	32	9
Mock	9	6	12	60

TABLE V: Migration segment extraction performance. P is the project number in our list. #KLOC is the number of kilo lines of Java code at the project latest version. #V is the number of the versions. T_{exact} and T_{ours} are resp. the extraction time in seconds of the *exact* approach and our approach. Γ_T is the time gain in percentage of T_{ours} with respect to the T_{exact} and Δ_S is the number of migration segments missed by our algorithm.

P	#KLOC	#V	T_{exact}	T_{our}	Γ_T	Δ_S
1	0.8	56	1.2	1.5	+25.7%	0
2	4.5	199	6.8	1.2	-82.4%	0
3	39	528	88	2.9	-96.7%	0
4	53	1095	442	7.5	-98.3%	0
5	1.1	106	2	0.2	-90.0%	0
6	2.9	76	3	0.5	-83.3%	0
7	2.9	56	2	0.3	-85.0%	0
8	3.4	116	10	0.8	-92.0%	0
9	29	4411	1643	13	-99.2%	0
10	9.1	453	30	3	-90.0%	0

performed in this paper. Binary search have been then executed within each segment to determine the exact bounds of the migrations. This value has been chosen thanks to our previous study on the subject [17] where we did not find a migration segment longer than 10 on a very large corpus of projects. We applied the algorithm described in Section III-B1 on our corpus to extract the migration segments corresponding to our migration rules. This operation took about three days of computation. The number of migration segments we extracted is shown in Table III.

2) Extracting the Migration Hunks: We retained 285 hunks from the migration segments. The number of functions we found in the hunks is shown in Table III. The number of software projects and the number of functions involved per migrations is also displayed. Only 36 migration segments were identified despite the large initial corpus of projects and the commonly used libraries. This reflects the difficulty to find software projects to apply our approach. The fact that library migration is an occasional practice constitutes the main impediment to the research of function mappings. Table IV shows the distribution of the hunks according to the number of functions they contain.

The number of functions involved within a hunk differ according to the migration rule. Indeed, all the hunks for the *I/O* migration rule have either 2 or 3 functions. This low value allows us to expect a good precision and many $1:1$ mappings. Inversely, the number of functions is greater for the *JSON* and *Mock* migration rules. Indeed, they have respectively 9 and 60 hunks containing between 7 and 10 functions. We expect the precision of our approach to suffer from such a number, and to extract many $n:m$ mappings.

3) Extracting the Function Mappings: To answer later RQ2, we decided to compare our approach with a similar technique proposed by Schäfer et al., that we will name method context [6]. Additional details will come in Section V-A2.

Two persons spent about one day to manually review the function mappings extracted using the two techniques. The two persons both practice Java development for respectively 4 and 8 years, but had no experience with the studied libraries. The *Javadoc* of the library functions and the textual *diffs* of the client project source code files were used as material for the review. Difficult cases were all discussed and decided in agreement. Our technique extracts 228 function mappings. Among them, we validated 115 function mappings and discarded 113 wrong ones. The precision of the approach is thus about 0.50%, which is fair. The union of the results of both techniques lead to a total of 135 correct rules that constitute our set of function mappings. The mappings extracted by our technique are available on-line on our Web page[2].

V. Results and Discussions

In this section we answer the four research questions listed in Section IV. Then, we discuss the threats to validity and limits of our approach.

A. Empirical Results

1) Segment Extraction Algorithm (RQ1): We compare our migration segment extraction algorithm against a technique we call *exact* that computes, for a given project p, the dependencies of each version $v_i \in V_p$. It then browses the history to identify migration segments for any migration rules $m \in M$. We call it *exact* since no library migration can be missed using this algorithm.

We applied the two algorithms on a similar corpus of 10 randomly selected projects where migrations were observed during our experiments. We measured the number of versions computed, the number of segments found and the running time. The time does not include the initial repositories cloning but only the segment extractions. The gains of our algorithm are also computed. In addition, we measured the average time required to extract the function within the detected migration segments. The results are displayed in Table V.

Three observations arise from these results. First, the two algorithm have detected a similar number of migration segments

[2]http://www.labri.fr/perso/cteyton/Matching/

TABLE VI: Precision and recall of the function mappings extracted by our approach (*hunk context*) and the *method context* approach of Schäfer et al. [6]. The recall is computed using the union of the correct mappings found by the two approaches.

	hunk context		**method context**	
Rule	#Correct	#Wrong	#Correct	#Wrong
I/O	21	1	18	10
Lang	40	7	38	30
JSON	29	64	25	163
Mock	25	41	11	42
Total	**115**	**113**	**92**	**245**
Precision	**0.50 %**		**0.27 %**	
Recall	**0.85 %**		**0.68 %**	

in these projects. After manual inspection, we found that the segments were associated to the same migrations. Our technique is thus able to detect migration segments. Second, the execution time is significantly lower and this clearly proves the efficiency and interest to use our algorithm to detect migration segments. Indeed, the time never reaches more than 13 seconds, and is on average around 80% faster than the *exact* algorithm. The value of 13 seconds in project #9 is simply due to a higher number of versions computed in this project compared to the others. Note that the computation time to extract the function rules is negligible so we do not enhance this point.

2) Migration Hunks vs. Methods (RQ2): To show that our technique based on hunks is relevant, we propose to compare the results we obtained with an implementation of the approach used by Schäfer et al [6]. In their work, they look for function replacements in a whole class method at once. Therefore we call their approach *method context* and ours *hunk context*. Within a segment and for a given method that appears in both versions of a file, we compute the set of added and removed method calls to functions of the libraries. Then, using a similar process to ours, we extract function mappings using the Cartesian product of added and removed methods. Note that methods are identified by their full signature. This approach is thus more robust to source code moves within a file, but inefficient when method names are changed.

The distribution of function mappings found per migration rule is shown in Table VI. We observe that the *method context* approach produced 337 function mappings, from which 92 were validated and 245 marked as false. The precision of this technique is 27% which is much less than ours. Our technique found 115 correct function mappings and 113 mappings were wrong. The proposed recall is computed according to the union of 135 correct rules extracted by both approach. The recall of our approach is significantly higher with our technique.

We observe that most of the validated mappings are part of *Lang*, as it contains the highest number of functions in its hunks. The precision is disparate since we obtain a very good value for *I/O* with only 1 wrong mapping. It confirms the expectation of the reported low number of functions in the hunks for this migration rule. The precision for the *Lang* mappings also reaches a good score with 40 correct mappings out of the 47 found. On the contrary, the precision is below 50% for the *JSON* and *Mock*. It confirms that the precision is impacted by the high number of functions in the hunks for this migration rule.

We explain the divergence of precision with the characteristics of library usage. Indeed, most of the mapped functions in *Lang* are static utility methods that have very specific purposes, and whose semantic is trivial to understand. Inversely, usage of *jmock* and *mockito* are quite different and reflects well the complexity of library migration. The manual inspection of the mappings for this migration rule was very tedious to perform. There were frequent functions from *jmock* that could not be mapped to any function from *mockito* or functions that could be mapped to many other functions.

We explain the differences of precision and recall between our approach and the *method context* approach by the nature of the *library update* problem addressed by Schäfer et al. Indeed, there is a small number of changes to detect between two versions of a library. Thus, considering the changes at the method level is sufficient. In our context, all the used functions of a library are totally replaced at once. As many library functions are often used in the same method, it produces a lot of false positives. Using hunks computed from textual differencing significantly reduces the number of false positives. The recall is also improved because of the ability to deal with renamed methods.

We show that our approach identified a fair number of function mappings. It also shows that analyzing hunks computed using textual differencing is a relevant solution. We obtain better results compared to the *method context* approach. The precision of the approach is hard to predict and depends on the library usage. If a typical library usage requires a combination of several methods, the cartesian product of the functions in the hunk is likely to generate many false mappings.

3) Textual Similarity and Multi-mappings (RQ3):

a) Function Multi-mappings: We now analyze the 135 function mappings to count the functions that are mapped to more than one function. Our goal is to understand if such a scenario exists in practice. In case it does not we could apply a greedy filtering technique that ensure mappings of cardinality 1 to 1. To obtain this result, we simply count how many functions are mapped to only one function (mono-mapped) and how many are mapped to more than one (multi-mapped). Table VII presents the results. It shows that the number of multi-mapped functions is stable for all migration rules: about 33% of the functions are multi-mapped. As it is a significant number, any 1 to 1 filtering technique will significantly decrease the recall.

b) Textual Similarity of Function Mappings: One motivation of our approach is that the textual similarity between the name of functions in the context of a library migration cannot be used. We use the set of 135 correct mapping to verify this hypothesis. To that extent, we compute the n-gram similarity [20] between the names of each mapped function. Then we filter out the function mappings for which the similarity is below a threshold t_n. Table VIII shows the effect

TABLE VII: Number of mono and multi mapped functions. A function is mono mapped if it appears in only one mapping. It is multi mapped if it appears in more than one mappings.

Rule	#Mono-mapped functions	#Multi-mapped functions
I/O	23	10
Lang	42	21
JSON	33	17
Mock	20	10

of this filtering on the number of discarded correct mappings. Our hypothesis is confirmed since only a minor subset of the function mappings are kept using textual similarity. *I/O* mappings have the most similar names, as our technique keeps 17 out of 23 mappings. However, the results on the other migration rules are poor. Assuming a t_n value of 0.5, only 27.1% of the *Lang* mappings are detected, 43.2% for *JSON* and 22.2% for *Mock*.

4) Filtering Technique Evaluation (RQ4): To improve the precision of our approach, we evaluate the filtering technique we described to assess if a satisfying trade-off between precision and recall is reached. We measure the impact of the t_{rel} threshold on the precision and recall of the extracted mappings. The recall is computed on the correct mappings detected right after the extraction from the hunks. We set t_{rel} to 10 values between 0 and 1. Table IX presents the different results obtained with these thresholds.

The filtering has not the same impact according to the migration rule. Indeed, for the *I/O* and *Lang* functions, it does not improve the precision of the results. On the contrary, it decreases the precision and recall. This is the case because the raw results of our approach were satisfying enough on these migration rules. On the other hand, the filtering has a positive impact on the *JSON* and *Mock* precisions. If we set $t_{rel} = 0.6$, the precision and recall reach respectively 81.8% and 62.1% for *JSON*. As expected, it also has the effect to reduce the recall. Moreover choosing the good threshold value seems hard because it depends on the migration rule. Nevertheless, this filtering step can be used when the precision of the results is poor to improve it.

B. Threats to Validity and Limits

Our approach has been tested on a subset of common Java migration rules. However, our corpus is not large enough for our results to be generalized on any migration rule. A larger corpus of libraries could be considered in a future experiment. The two persons that performed the manual review might have made mistakes when evaluating the correctness of the mappings. We performed a comparison of our migration segment extraction technique on only 10 projects with an *exact* technique, because of its long computation time. Therefore we cannot generalize the efficiency of our algorithm.

TABLE VIII: Evaluation of syntactic similarity for the function mappings.

t_n	**I/O**	**Lang**	**JSON**	**Mock**
0.5	82,6%	27,1%	43,2%	22,2%
0.6	73,9%	16,7%	37,8%	18,5%
0.7	73,9%	12,5%	29,7%	18,5%
0.8	73,9%	12,5%	27%	18,5%
0.9	73,9%	12,5%	27%	18,5%
1.0	73,9%	12,5%	27%	18,5%

We can not discuss situations where our technique misses to detect migration segments as we did not encounter one, but the drawbacks of our algorithm are explained in Section III-B1. Also, we did not manually verified the migration segments to assess their correctness. Our mapping extraction technique based on the hunks fails when functions are extensively modified. In this case many false positives are generated. Our technique can also fail if the replaced functions are exactly the same in the old and new libraries, as we rely on textual differencing. While we did not compute the recall, as manually finding the mappings between all the functions of two libraries is very long and difficult, it is obvious to see that it highly depends on the number of projects that already performed the migration. The more projects there are, the more it is likely to see how a function is replaced. If only few projects using few functions performed the migration, our approach fails to extract mappings.

Our technique analyzes each project version recorded on their repositories. This level of granularity may be too low since bugs can be introduced during software development, and thus the quality of the code analyzed is not always guaranteed. This aspect has to more discussed in a future work.

VI. Conclusion and Future work

In this paper, we tackle the challenge of assisting library migrations. We propose to ease this process by automatically extracting function mappings from the analysis of software projects that already performed the library migration.

Our contributions are the following. We propose an efficient algorithm to identify migration segment through the set of versions of a project. A migration segment is the shortest interval where a migration can be observed. Within such segments, we extract the source code transformations involved in the migration by analyzing hunks containing removed call of functions of the old library, and added calls to functions of the new library. These hunks are obtained using textual differencing between the old and new version of a modified file. The removed and added functions of such hunks are likely to be equivalent. This operation extracts functions mappings. We provide a filtering technique based on relative similarities to reduce the number of false positives. We deployed a large-scale analysis of Java open source projects to evaluate our approach in practice.

The conclusions of our survey are as follows. First, we show that our approach is promising since function mappings are identified for a small set of common Java migration rules. Second, we show the efficiency of our migration segment extraction technique, which is significantly faster than an exact approach while having similar results. Third, the hunk-based function mappings extraction works well in the context of

TABLE IX: Correct rules detected and precision of the approach with severals t_{rel} values.

	I/O		**Lang**		**JSON**		**Mock**	
t_{rel}	Prec	Rec	Prec	Rec	Prec	Rec	Prec	Rec
0.0	*95.5%*	*100%*	*85.1%*	*100%*	*31.2%*	*100%*	*37.9%*	*100%*
0.2	95,5%	100%	84,8%	97,5%	39,1%	93,1%	46,4%	52%
0.4	94,7%	85,7%	83,7%	90%	61,9%	89,7%	64,7%	44%
0.6	93,3%	66,7%	82,4%	70%	81,8%	62,1%	60%	24%
0.8	92,9%	61,9%	81,8%	67,5%	81,8%	62,1%	50%	16%
1.0	92,9%	61,9%	79,3%	57,5%	81%	58,6%	50%	8%

library migration. It generates mappings with a very good precision in two cases out of four. Finally, our filtering technique can improve the precision in the cases where the initial precision is bad.

In a future work, a study in collaboration with real developers could help us find the most suitable format to report the function mappings. In addition, we are willing to replicate our study on larger open source and industrial systems so that the results can be generalized. We also want to investigate several hunk and function mappings filtering techniques, such as computing function bodies similarity. Finally, we want to evaluate the effect of our approach on the development effort and quality by asking developers to perform migrations with and without our mappings.

REFERENCES

[1] M. T. Baldassarre, A. Bianchi, D. Caivano, and G. Visaggio, "An industrial case study on reuse oriented development," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ser. ICSM '05. Washington, DC, USA: IEEE Computer Society, 2005, p. 283–292.

[2] C. Teyton, J.-R. Falleri, and X. Blanc, "Mining library migration graphs," in *19th Working Conference on Reverse Engineering 2012, 15th-18th October 2012, Kingston, Ontario, Canada*, IEEE, Ed., Kingston, Ontario, Canada, Oct. 2012, pp. 289–298.

[3] E. Duala-Ekoko and M. P. Robillard, "Asking and answering questions about unfamiliar APIs: an exploratory study," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, p. 266–276.

[4] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *Proceedings of the 20th European conference on Object-Oriented Programming*, ser. ECOOP'06. Berlin, Heidelberg: Springer-Verlag, 2006, p. 404–428.

[5] 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, p. 325–334.

[6] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *Proceedings of the 13th international conference on Software engineering - ICSE '08*, 2008, p. 471.

[7] Z. Xing and E. Stroulia, "UMLDiff: an algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, p. 54–65.

[8] B. Dagenais and M. Robillard, "SemDiff: analysis and recommendation support for API evolution," in *IEEE 31st International Conference on Software Engineering, 2009. ICSE 2009*, 2009, pp. 599–602.

[9] 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," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, p. 302–321.

[10] M. Kim, D. Notkin, and D. Grossman, "Automatic inference of structural changes for matching across program versions," in *Proceedings of the 29th international conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, p. 333–343.

[11] S. Kim, K. Pan, and E. J. Whitehead,Jr., "When functions change their names: Automatic detection of origin relationships," in *Proceedings of the 12th Working Conference on Reverse Engineering*, ser. WCRE '05. Washington, DC, USA: IEEE Computer Society, 2005, p. 143–152.

[12] P. Weissgerber and S. Diehl, "Identifying refactorings from source-code changes," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '06. Washington, DC, USA: IEEE Computer Society, 2006, p. 231–240.

[13] Z. Xing and E. Stroulia, "API-Evolution support with diff-CatchUp," *IEEE Trans. Softw. Eng.*, vol. 33, no. 12, p. 818–836, Dec. 2007.

[14] B. E. Cossette and R. J. Walker, "Seeking the ground truth: a retroactive study on the evolution and migration of software libraries," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, p. 55:1–55:11.

[15] T. Tonelli Bartolomei, K. Czarnecki, R. Lämmel, and T. v. d. Storm, "Study of an API migration for two XML APIs," in *2nd International Conference on Software Language Engineering (SLE)*, vol. 5969/2010, Denver, USA, Oct. 2009, pp. 42–61.

[16] 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, Sep. 2010.

[17] C. Teyton, J.-R. Falleri, M. Palyart, and X. Blanc, *A Study of Library Migration in Java Software*, 2013.

[18] E. W. Myers, "An o(ND) difference algorithm and its variations." in *Algorithmica*, 1986, pp. 251–266.

[19] S. Melnik, H. Garcia-Molina, and E. Rahm, "Similarity flooding: A versatile graph matching algorithm and its application to schema matching," in *ICDE*, 2002, pp. 117–128.

[20] C. E. Shannon, "A mathematical theory of communication," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 5, no. 1, p. 3–55, Jan. 2001.

[21] S. Okur and D. Dig, "How do developers use parallel libraries?" in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, p. 54:1–54:11.

[22] S. Thummalapenta and T. Xie, "SpotWeb: detecting framework hotspots and coldspots via mining open source code on the web," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, p. 327–336.