Automating the Detection of Third-Party Java Library Migration At The Function Level

Hussein Alrubaye, and Mohamed Wiem Mkaouer
Rochester Institute of Technology
Rochester, New York
{hat662,mwmvse}@rit.edu

ABSTRACT

The process of migrating between different third-party libraries is very complex. Typically, developers need to find functions in the new library that are most adequate in replacing the functions of the retired library. This process is subjective and time-consuming as developers need to fully understand the documentation of both libraries' Application Programming Interfaces, and find the right match between their functions if it exists. In this context, several studies rely on mining existing library migrations to provide developers with by-example approaches for similar scenarios. In this paper, we propose a mining approach that extracts all the manually-performed function replacements for a given library migration. Our approach combines the mined function-change patterns with function-related lexical similarity to accurately detect mappings between replacing/replaced functions. Using our enhanced mining process, we perform a comparative study between state-of-art approaches for detecting migration traces at the function level. Our findings have shown its efficiency in accurately detecting migration fragments and it has enhanced the accuracy of state-of-art approaches in finding correct functions changes. We finally provide the community with a dataset of migrations between popular Java libraries, and their corresponding code changes at the function level.

ACM Reference Format:
Hussein Alrubaye, and Mohamed Wiem Mkaouer. 1997. Automating the Detection of Third-Party Java Library Migration At The Function Level. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering (CASCON'18).* ACM, New York, NY, USA, Article 4, 12 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Software systems heavily rely on the functionality of the third-party library as a mean to save time, reduce implementation cost, and increase quality when offering robust services. On the other hand, using third-party code has its own challenges, mainly related to its maintenance and evolution since using outdated libraries augments the bug proneness of the code; furthermore, with the explosion of mobile applications, using inappropriate libraries increases the attack surface and security vulnerabilities [2]. Thus, as software systems evolve, the need for better services and more secure and reliable functionalities causes developers to often replace one library with another library. This process of replacing one library with a completely different library, while preserving the same functionality, is known as library migration [15].

During the migration process, developers are required to find the right replacing function(s) for each removed function belonging to the retired library. Developers are also required to verify whether the newly introduced functions are delivering the same expected functionalities of the retired library. These tasks tend to be subjective, time-consuming, and error-prone, as developers need to fully understand how both libraries function and be aware of their structures. This may include exploring their documentation and searching online for practical examples of their usages. Moreover, the matching process between the replacing and replaced functions, belonging respectively to the retiring and retired libraries, is not straightforward. Even if libraries offer similar functionalities, they eventually differ in their design and documentation [1].

Library migration differs from library upgrade, since the latter is conceived as a migration between two different releases of the same library. Both library upgrade and migration have been attracting several research studies that focus on the evolution of third-party libraries in software systems. They mainly analyze the challenges that software engineers face as libraries evolve, in general, to reduce the cost of their maintainability by preserving their backward compatibility and preventing them from introducing malfunctions and breaking changes. Even if library migration is considered part of library evolution, it differs from the upgrade since developers have made the decision of retiring a library and replacing it with a syntactically and structurally different one, and they are ready to apply the necessary code changes to support the migration.

The purpose of our research is to help the developer better comprehend how the migration process is to be fulfilled, by providing the software engineering community with real-world migration examples, extracted from a wide set open source projects. However, the detection of an existing migration is challenging. There is no systematic way to detect the developers' intention of applying a migration without them explicitly performing it at the source code level. Thus, the detection of library migration requires extensive analysis of the history of code changes while searching for specific patterns. Furthermore, in contrast with library upgrade, the migration process may be applied on one or multiple series of code changes, as developers are removing and adding functions that may be distributed throughout several source files of the project. Since these migration-related changes can be easily interleaved with other code changes, distinguishing them becomes challenging.

Existing studies [5, 11, 12] have been focusing on searching for the most adequate replacing library for a given one to retire.

CASCON'18, October 2018, Markham, Canada

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

Approaches recommend libraries to migrate to based on several features linked to semantic similarity, API documentation similarity, identical functionalities, common hosting projects etc. However, this recommendation is limited to the library level without recommending the necessary changes at the function level. Thus, the need of automating the migration has attracted researchers to mine existing migrations in order to understand how they are performed and to extract the features that can be utilized to automate the recommendation of libraries at the function level [4, 5, 13, 15]. The main features used to approximate the migration traces at the function level rely on (1) identifying frequent change patterns between replacing/replaced functions, (2) calculating the similarity between functions based on their signatures, internal or external documentation, and (3) detecting the function's similar usage context(s). However, these features heavily rely on correctly identifying migration fragments in the source code, which is challenging to mine. Based on our study, developers do not necessarily explicitly mention their intention to migrate libraries, and when they do, they generally announce the migration between libraries without documenting all the code changes made at the function level. Moreover, there is no existing datasets of migrations between libraries to support the learning and automation of the migration.

To cope with the above-mentioned challenges, this paper identifies the migration traces between a pair of given libraries. It takes as input a pair of libraries and extracts the developer's decisions with respect to changes at the function level. Each removed function belonging to the replaced library is mapped, if detected, to one or multiple added functions, belonging to the replacing library. To do so, we propose to extend the study of Teyton et al. [16] as follows:

- We enhance the existing mining approach to increase the accuracy of detecting migration fragments in the period of migration.
- We compare the function mapping generator algorithm proposed in the previous study with two other state-of-art approaches that we adapted for this problem.
- We extend the number of studied migration pairs from 4 to 12 and we extend the number of analyzed projects when searching for migrations from 12,000 to 57,447 projects.
- We perform a comparative study between state-of-art approaches that identify the migrations at the function-level.
- We provide the generated migration results as a dataset for the research community to better comprehend how developers achieve this practice.

The results have shown that our proposed mining algorithm reveals a higher number of migration traces on the source code level. This allows us to increase the performance of state of the art algorithms in terms of accurately detecting migration traces. We also found that there is no approach that outperforms all the other approaches in all scenarios.

The paper is structured as follows: Section 2 presents the terminology that is used throughout the paper. Section 8 enumerates the studies relevant to our problem. Section 3 explains the challenges related to extracting existing library migrations. The mining approach is detailed in Section 4, it also issues an example to illustrate how each of the approaches under comparison generates the mappings at the function level. Section 5 shows our experimental methodology in collecting the necessary data for the experiments that are discussed in Section 6, followed by the conclusion and future directions in Section 9.

2 BACKGROUND

This section presents definitions of keywords that are used throughout this paper.

Library. A library encapsulates a set of resources, in the form of objects and functions, publicly accessible through the library's Application Programming Interface (API). Just like any traditional software, a library has multiple releases. Note that in this study, we identify libraries by the composition of their GroupID, ArtifacID and version, but since we are interested in the in-between library evolution, we label libraries by their artifactID for the sake of simplicity.

Library Migration. A migration process occurs when a target library is replacing a source library. The source library is considered retired if all of functions dependencies are removed from the software. This definition does not require the source library to be physically removed from the project, it just enforces that none of its functions are actually used to be considered a migration.

Migration Rule. the pair of a source (retired) library and a target (replacing) library. (*jmock* to *mockito*) is an example of a migration rule. Table 4 has the list of all migration rules considered in this study.

Function Mapping. A migration rule is a set of function mappings between the source and the target library. The mapping between functions is the process of replacing a least one function from the source library by one or multiple functions belonging to the target library. Fig 5 contains some examples of function mappings that are generated using different approaches.

Fragment. A code block witnessing at least one function mapping. It is generated by contrasting the code before and after the migration to only keep the code block containing the removed (resp.added) functions linked to the source (resp. target) library. Fig 5 contains three fragments, each fragment contains a set of added/removed functions.

Segment. It constitutes the migration *period*. It is a sequence of one or multiple code changes (e.g., commits), containing one or multiple fragments. A segment can be constituted by at least one commit, if the migration occurs instantly, or it can be propagated through several commits, if the migration was gradually performed over time. For example, in Figure 1, commit number 5 has contained all function mappings between the removed *json* and the added *gson*, so it is considered the segment of that migration.

The next section applies this terminology when defining the problem of mining migration between libraries.

3 PROBLEM STATEMENT

The problem of inferring function mappings between two given libraries raises many challenges related to the following scenarios:

Types of migration. During our analysis of the mined migrations, we noticed mainly two migration patterns: We label the first pattern *instant migration*. It occurs when developers add the target library, perform all function replacements within the same code revision unit, e.g., commit. Taking the examplf of *ps2-parser* project

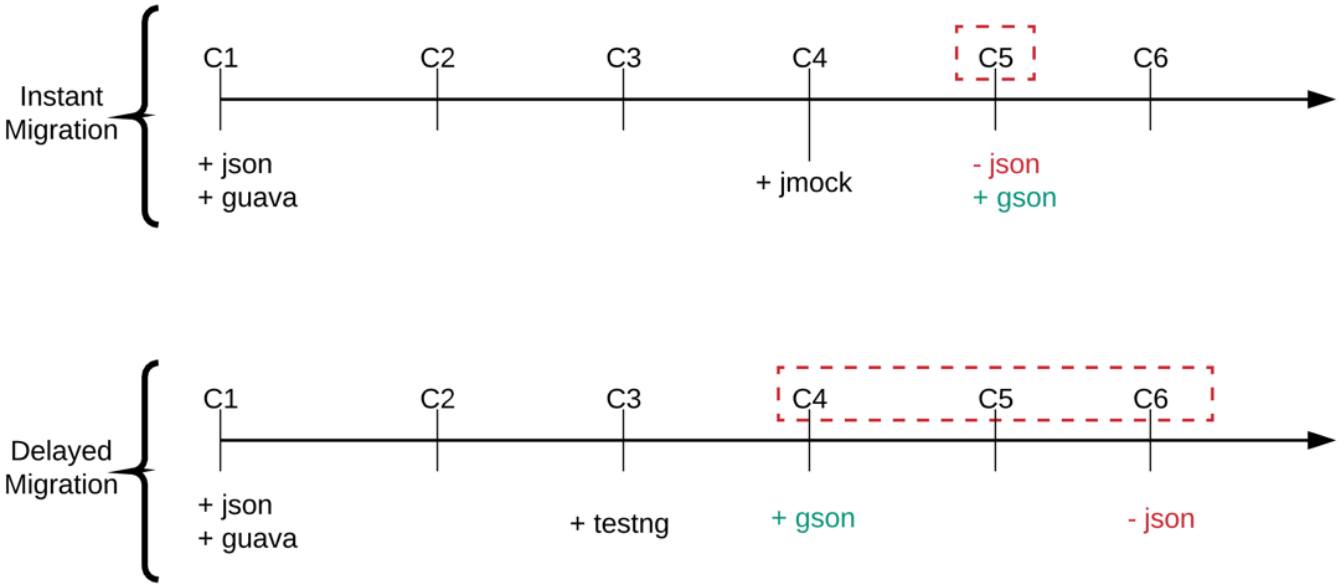

Figure 1: Example of segments in instant and delayed migrations.

[1], the *junit* library was removed, and the *testng* library was added in the same commit. The second pattern is denoted *delayed migration*. In this case, developers add the target library, periodically perform function replacements through multiple revision units, until the project is no longer dependent on the source library. Obviously, the first pattern is straightforward for extracting migration fragments to be mined. But interestingly, the second pattern tends to be popular since developers may opt to gradually perform these changes while observing the system's behavior. In the *redmine-java-api* project [2], *json* has replaced *gson*, but the latter was not removed from the project.

To handle this challenge we extended the mining algorithm proposed by Teyton et al. [15, 16] by analyzing each project at the revision unit level instead of the version-level, i.e., we compare all pairs of project commits when searching for potential fragments instead of comparing between two *version* commits, in which, the latter one has all in-between commits merged.

Multiple function replacements. Ideally, every source library function is replaced with at most one target library function, and their function signature is identical. In reality, due to the difference in libraries design and vocabulary, we observed several situations where one function is being replaced with more than one function. In this example[3], on line 122, the method *put(key, value)* has been replaced with two methods, namely *addProperty(key, new Gson().toJson(value)*.

To address this challenge we implemented 3 mapping generation algorithms [11, 14, 16] that are evaluated in terms of their accuracy later in the experiments. To ensure the comparison fairness, we provide the same fragments for each migration rule as input to each algorithm.

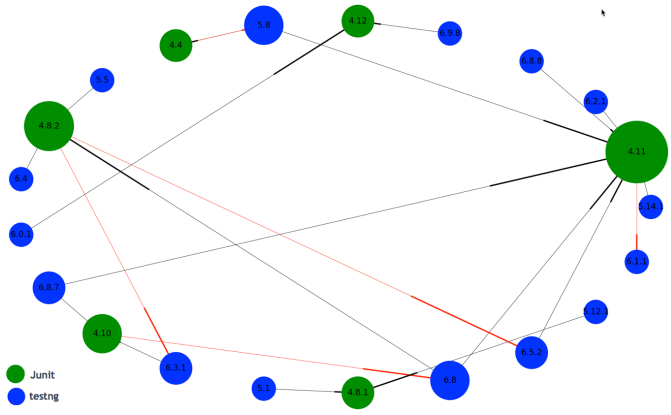

Figure 2: Several migrations between testng and Junit. occuring between various releases

4 APPROACH

In this section, we decompose our research methodology into three main phases: (1) Detection, (2) Mapping, and (3) Validation.

4.1 Detection.

In this phase, we input a list of open source Java projects. We clone and check out all commits for all projects. For every commit, we collect its properties, such as commit ID, commit date, developer name, and commit text. We also keep track of all changes in the project library configuration file, known as Project Object Model

[1]https://goo.gl/Svy2o2
[2]https://goo.gl/KADNoK
[3]https://goo.gl/febhX3

(*pom.xml*). All mined project data is saved in a database for faster querying when identifying of segments and fragments.

Segment Detection. The purpose of our next search is to locate for each migration rule, its migration segments in all projects. As defined in the background section, a segment is composed of one or many commits involved in the migration process by containing fragments. As shown in Figure 3, the *Segment Detector* starts by checking whether both libraries exist in the list of added/removed project libraries. It locates the end of the segment by scanning all commits while looking for any commit in which all project source files are no longer dependent on the library to retire. It performs this deep scan because our migration definition does not require physically removing the library from the project to be considered retired. Once the segment end is located, we keep scanning previous commits in a backward fashion, looking for the beginning commit which contains the first fragment i.e., first replacement of any retired library function. After locating all segments for a given migration rule, it is critical to keep track of the source and target library versions for each segment to avoid the false positive detection of an API change between two versions of the same library as a migration. Figure 2 reveals all recorded migrations between several versions of *testng* and *Junit*. Note that this migration is symmetric, i.e., there is a migration going from *testng* to *Junit* and vice versa. However, for a given migration rule, only segments from the source to the target are considered.

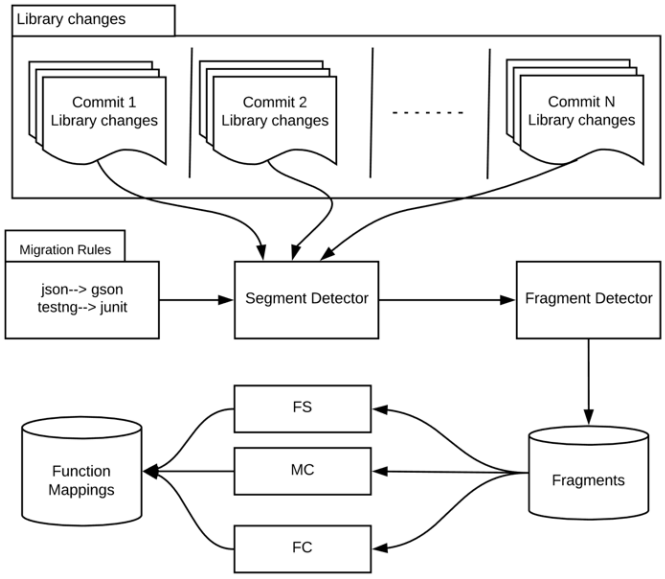

Figure 3: Overview of the Automated function mapping generation.

Fragment Detection. The *Fragment Detector* in Figure 3 is responsible for the fragment extraction. It clones the project source files that are changed in the commits belonging to the segment. It applies the *Unified Diff Utility* command between the files that changed to generate fragments. A fragment is a continuous set of lines that have been changed along with contextual unchanged lines. Only fragments containing removed (resp. added) functions from the source (resp. target) library are considered valid.

The fine-grained comparison between commit pairs is computationally expensive and becomes challenging for projects containing a large set of commits. Fortunately, these comparisons are independent and so they can be parallelized.

Table 1: Illustrative example of Two libraries, and their functions.

Library	Functions
json	JSONParser() String toJSONString() long getLong(String) String get(int) void add(String)
gson	JsonParser() String toString() long getAsLong() JsonElement get(String) void add(String, JsonElement)

Table 2: List of Fragments detected for (*jmock →mockito*).

Fragment	Added/Removed Function
1	- String toJSONString() - String get(int) - void add(String) + String toString() + JsonElement get(String) + void add(String, JsonElement)
2	- String toJSONString() + String toString()
3	- String get(int) + long getAsLong()

As an illustrative example, in Figure 4, we run *diff* between two versions of class *Eval.java*[4] to generate *Diff Eval.java*. Since the fragment is composed of interleaved changed and unchanged code, we discard any lines of unchanged code unrelated to the migration. We extract the function signature for every method call in the fragment, using the Abstract Syntax Tree (AST) of the library classes and methods signatures, as shown in Figure 4 (D). We only keep fragments containing at least one function call from each source and (target library. Once fragments are sanitized from unrelated changes, they are transferred to the next phase of function mapping generation. Since we are using a larger set of projects, in comparison with the previous study [16], We retained a total of 8,938 fragments from the migration segments while the previous study retained 285 fragments.

[4]https://goo.gl/mqsshW

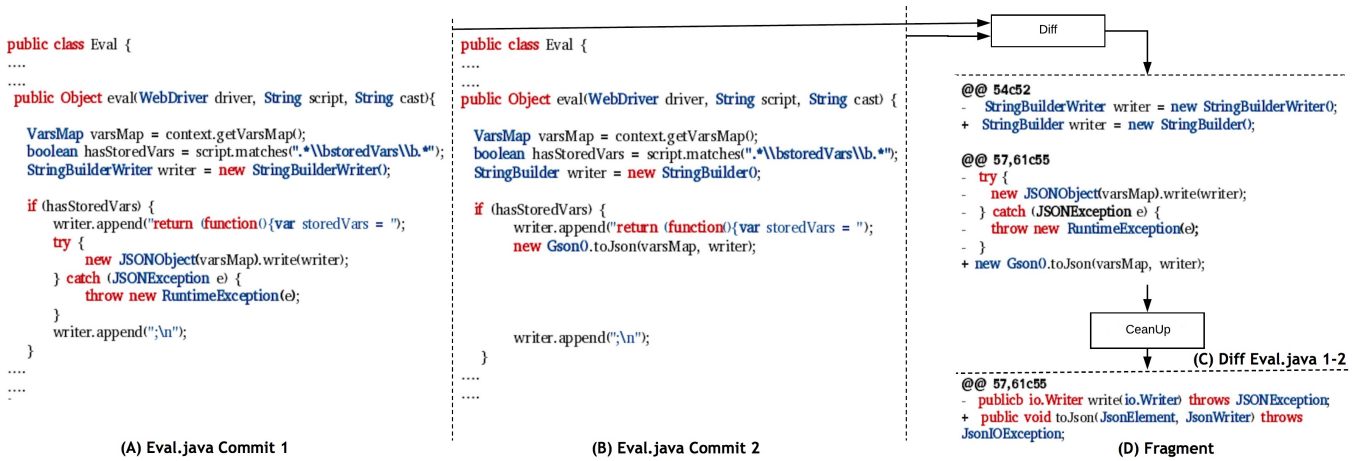

Figure 4: Fragment detection in *Eval.java* from *json* →*gson*.

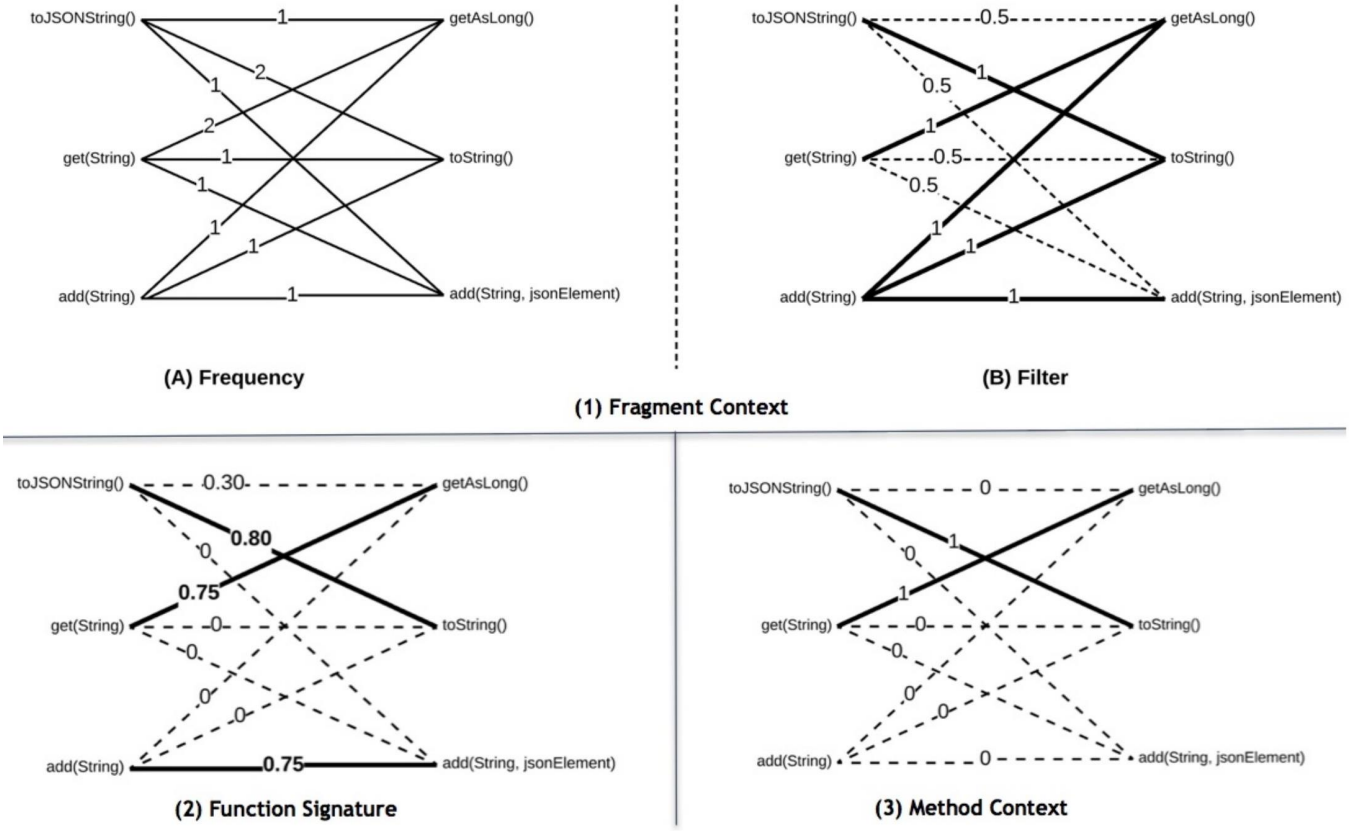

Figure 5: Function mapping generation for the *json* →*gson* migration rule, the functions are provided in Table 1, and the migration fragments are provided in Table 2.

4.2 Mapping.

The purpose of this phase is to produce mappings between functions for given input fragments. A high-level overview of the process is described in Figure 3.

As shown in Figure 5, we feed the same set of fragments for the three algorithms. For the sake of simplicity, we label the approach of Teyton et al. [16] *Fragment Context (FC)*. Similarly, we label the

approach of Schäfer et al. [14] and Nguyen et al. [11] respectively *Function Context (MC)* and *Function Signature (FS)*.

Fragment Context (FC). To extract all the possible combinations between the set of added and removed functions the Cartesian Product (CP) is performed between the set of functions in the fragments. Figure 5.1 (A) demonstrates the CP process in the form of a graph. Every node in the graph represents a function while the edge represents its corresponding mapping to another function. For instance, the edge between *toJSONString()* to *toString()* has a weight of two because this mapping exists in the first two fragments. Since the CP generates every possible combination of mappings, its results contain a large number of false positives. Thus, a filtering process [10] is performed: as shown in Figure 5.1 (B), the weights are normalized by the highest outgoing weight per node, then the only mappings kept are those with a normalized weight that is higher than a user defined filtering threshold value $t_{rel} \in [0, 1]$. The value of t_{rel} controls the selection strictness. For example, when the filter $t_{rel} = 1$, the *toJSONString()* to *toString()*, *get(String)* to *getAsLong()*, and *add(String)* to (*getAsLong(String),toString(),add(String,jsonElement)*) mappings are selected.

Function Signature (FS). This approach calculates the function signature similarity for each combination of functions as follows [11]:

$$\begin{aligned} FS(fct1, fct2) = {} & 0.25 sm(fct1\_\{returnType\}, fct2\_\{returnType\}) \\ & + 0.5\, lcs(fct1\_\{name\}, fct2\_\{name\}) \\ & + 0.25\, lcs(fct1\_\{param\}, fct2\_\{param\}) \end{aligned} \quad (1)$$

where *sm()* calculates the token-level similarity [8] between the two return types and *lcs()* computes the longest common subsequence between the two given input function names [6]. Figure 5.2 plots the graph of functions pairs with their associated similarity values. Finally, for each source function, only target function(s) with the highest similarity score is kept. As shown in Figure 5.2, three mappings were generated.

Method Context (MC). Given a set of fragments as input, this approach considers only one-to-one function mapping with the highest edge weight. Figure 5.3 is the generated graph. Note that the edge between *toJSONString()* to *toString()* and *get(String)* to *getAsLong()* has the weight of 1 as each of these functions exists on a separate fragment. This approach generates 2 function mappings from *toJSONString()* to *toString()* and from *get(String)* to *getAsLong()*.

We observe from the previous illustrative example that the 3 approaches give different results. This motivates us to perform a comparative study which is detailed in the following section.

4.3 Validation.

To validate the function mappings generated by three approaches, we conducted the manual validation process similar to the validation process in the previous study [16]. We performed the manual validation of unique mappings that are generated by the three approaches by building a web portal [5] that shows every function mapping with a list of the projects' commits that contain each mapping when found. The authors then decide the correctness of the rule by verifying the function mapping in the list of commits. For example, line 55 in this commit [6] has a valid mapping between *toJSONString()* and *toString()*. The number of mappings generated by the three approaches is shown in Table 4. Also, the mined migrations, along with their manual validation are available in the above-mentioned web portal for replication and extension purposes. The manually verified mappings constitute the ground-truth that we use in our experiments when conducting the comparative study between state-of-art mapping generation approaches.

5 EXPERIMENTAL DESIGN

We design our experiments to mainly compare the three function mapping approaches, namely *FC*, *MC*, and *FS*. Then, we investigate the impact of t_{rel} on the performance of *FC*. Lastly, we evaluate our commit-based extraction approach in terms of its accuracy in revealing existing migrations. We design our methodology through the definition of the following research questions.

Research Methodology. Our study aims at addressing the following research questions outlined below:

- **RQ1 (Comparison).** How do the 3 algorithms perform in the generation of correct function mappings?
- **RQ2 (Tuning).** How does the value of the filter t_{rel} impact the performance of the Fragment Context approach?
- **RQ3 (Efficiency).** To what extent can commit-based extraction identify relevant fragments for a given migration rule?

To answer the RQ1, we conduct a comparative study of the approaches. Since 2 of the 3 approaches rely on the frequency of detected mappings in deciding on their validity, it is important for our experiment to provide them with a large sample of fragments. For this purpose, we mined 57,447 Java projects. Furthermore, we varied the rules from 4 to 12 to challenge the algorithms with a diverse set of libraries. These migration rules were manually validated and provided by Teyton et al. [17]. As for RQ2, we measure the performance of *FC* while varying the values of t_{rel}. Answering the RQ3 is two-fold, we first verify the correctness of our mining algorithm that we label the *commit-based* approach by its ability to detect a set of manually verified fragments. Second, we compare it with the original mining algorithm of the previous study which we label the *version-based* approach. In addressing the RQs, the following metrics are used:

Precision. It denotes the ratio of correct extracted function mappings of all generated mappings.

$$Precision(x) = \frac{Vx}{Vx + Ix}$$

where Vx the total number of valid mappings that are found by an algorithm and Ix is the total invalid mapping that is found by an algorithm.

Recall. It denotes the ratio of correct extracted function mappings of all expected mappings.

$$Recall(x) = \frac{Vx}{Ox}$$

[5] http://migrationlab.net/index.php?cf=cascon2018

[6] https://goo.gl/kWnLk8

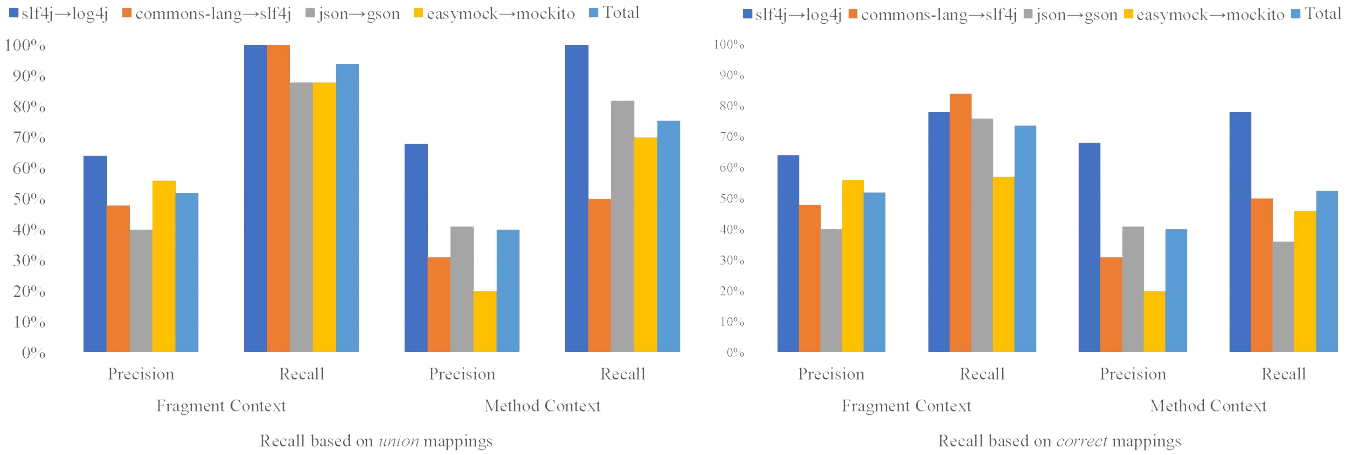

Figure 6: Precision and recall using *union* and *correct* mappings.

where O denotes the group of expected mappings, if ($O = C$) then the group of expected mappings is composed of all, manually verified, *(C)orrect* mappings. If ($O = U$) then the set of expected mappings is a subset of C, containing only the *(U)nion* of the correct mappings found by the algorithms under comparison.

6 RESULTS

Results for RQ1. We applied the *FC*, *MC*, and *FS* approaches on our extended set of 8,938 fragments. We then calculated the precision and recall based on the ability of each approach to replicate the manually validated mappings. Since the *FC* approach relies on user-defined filtering threshold t_{rel}, it is important to tune it with respect to each input i.e., migration rule. Thus, for the fairness of the comparison, we measure the performance of *FC* in terms of precision and recall while varying t_{rel} between 0 and 1, for each migration rule. Then, we select the filtering value of t_{rel} for which *FC* has the highest accuracy, per migration rule. The complete list of accuracies for all migration rules is available online[7].

Table 3: Average Precision and recall of FC and MC using *union* mappings.

	Union (U)			
	FC		MC	
	Precision	Recall	Precision	Recall
Our study	52%	94%	40%	76%
Teyton et al. [16]	50%	85%	27%	68%

Figure 6 replicates the previous study's comparison of *FC* with *MC* when generating migrations for the 4 rules used in the previous experiment [16]. They are initially evaluated using the *union* of correct mappings already found by *FC* and *MC*. We extend Figure 6 to include their performance in generating the manually validated mappings that we denoted *correct*. We also compare our findings in terms of the average precision and recall with the previous study in Table 3. We observe that we were able to reproduce relatively similar but relatively higher precision for *FC* which was around 50% in the previous study while it is 52% in our study. Also, *FC* scored a recall of 94%, 9% higher than the previous study. This is an indicator of the closeness of our algorithms in reproducing the previous study results. We notice a general increase of precision and recall values for all the approaches, as shown in Table 3, as a consequence of diversifying the set of fragments, and so allowing both algorithms to accurately detect repetitive mappings. Similarly to the previous study, we also notice, in both scenarios, that *FC* outperformed *MC* in terms of precision and recall, but its recall has drastically decreased when computed using *correct* mappings. This is due to the nature of *union* that may miss several correct mappings, not being detected by either of the approaches. In this case, we manually checked that *FC* correctly identified 192 out of 314 correct function mappings.

As we align our findings with the previous paper's since *FC* has outperformed *MC*, we extend the comparison to include *FS* in the comparison. Table 5 reports the precision and recall of the 3 algorithms. Interestingly, *FS* generated a comparable set of correct mappings across all migrations with a significantly fewer set of wrong mappings, when compared with the wrong set of *FC*.

Table 5 shows that *FC*, *MC* and *FS* performed differently depending on the migration rules. First, we notice that the number of functions per-fragment (or fragment density) has a high effect on *FC* and *MC*. When a migration rule is dominated by one-to-one mappings, it significantly increases the precision and recall of *FC* since the Cartesian product will generate only one mapping, which is correct. Similarly, for *MC*, the existence of one added and one removed function per fragment represents its best case scenario which explains its high precision and recall as well.

For example, for the migration rule *google -collections→guava* , the recall for both approaches is 78%. Table 4 reveals that all the

[7]http://migrationlab.net/index.php?cf=cascon2018

Table 4: Mined migration rules, their number of unique segments, fragments, function mappings, and functions per-fragment across all projects.

	# of (Segments/Fragments/Mappings)			# of Functions Per Fragment				
Rule (Source →Target)	**Segment**	**Fragment**	**Mapping**	**0-2**	**3**	**4-6**	**7-10**	**>10**
commons-logging→slf4j	102	4037	54	3975	19	23	11	9
easymock→mockito	65	2643	52	1300	725	514	88	16
testng→junit	31	963	71	947	6	9	1	0
slf4j→log4j	11	101	14	98	3	0	0	0
json→gson	12	75	30	64	5	5	0	1
json-simple→gson	4	49	10	42	3	4	0	0
google-collections→guava	8	39	19	39	0	0	0	0
jsf→javax.faces	4	35	5	34	0	0	0	0
gson→jackson	17	32	24	28	1	1	1	1
commons-lang→slf4j	12	28	8	21	4	1	0	2
sesame→rdf4j	1	18	4	16	1	1	0	0
jets3t→aws-sdk	2	7	12	5	1	0	0	1

Table 5: Precision and recall of the function mappings. The recall is computed using *correct* mappings.

	Fragment Context			Method Context			Function Signature		
Rule (Source →Target)	**Prec.**	**Rec.**	**Ac.**	**Prec.**	**Rec.**	**Ac.**	**Prec.**	**Rec.**	**Ac.**
commons-logging→slf4j	46%	35%	25%	20%	29%	13%	89%	31%	30%
easymock→mockito	56%	57%	44%	20%	46%	19%	76%	20%	33%
testng→junit	52%	56%	37%	34%	49%	25%	90%	76%	63%
slf4j→log4j	64%	78%	55%	68%	73%	57%	72%	57%	47%
json→gson	30%	76%	27%	31%	36%	20%	68%	43%	36%
json-simple→gson	40%	70%	33%	26%	40%	19%	62%	50%	38%
google-collections→guava	60%	78%	51%	83%	78%	68%	82%	73%	63%
jsf→javax.faces	21%	100%	21%	66%	80%	57%	71%	100%	71%
gson→jackson	28%	50%	22%	42%	37%	25%	81%	54%	48%
commons-lang→slf4j	48%	100%	46%	31%	50%	27%	27%	62%	23%
sesame→rdf4j	57%	100%	57%	57%	100%	57%	100%	88%	80%
jets3t→aws-sdk	20%	75%	19%	28%	16%	11%	80%	83%	83%
Total	47%	**73%**	31%	45%	52%	31%	**74%**	61%	**55%**

39 fragments of this rule are dominated by one-to-one and one-to-two mappings. On the other hand, considering the rule *commons-logging→slf4j*, which has a larger number of n-to-m mappings, per fragment, we realize that *FC* (resp. *MC*) precision and recall are lower than 64%, even with only 6% of fragments, having more than 1-to-2 mappings per-fragment. Yet, this has no observed effect on *FS*.

FS has uniform precision and recall percentages across all the migration rules that we analyzed. Interestingly, *FS*'s accuracy of 55% was the highest between all approaches, on average. It is important to note that *FS*'s performance didn't decrease when the density of fragments increases. This means that the function signature similarity is a good measurement to distinguish pairs of mappings on a tangled migration. Theoretically, this approach suffers in case of discrepancy of vocabulary used to describe API functions, which is practically rare, at least for the set of popular libraries that we are investigating. On the other hand, *FS* scored a precision of 100% for detecting mappings of the rule *sesame→rdf4j*. A closer look at this rule shows that *sesame* is the predecessor of *rdf4j*, as shown in Table 4, there are only 4 mappings identified, and these mappings link functions with identical signatures from both libraries, which explains the optimal performance of *FS*. In general, *FS*'s good precision infers the high similarity in the naming and vocabulary of libraries providing similar functionalities, which explains the heavy reliance on lexical similarity as a fitness for the problem of recommending similar libraries [12, 13].

In contrast with the previous study, we conclude that *FC* does not outperforms the other approaches across all libraries. Although, *FC* has the highest recall value of 73%, *FS* has the best precision value of 74%. Thus, the decision of championing an approach over another is subjective. Considering the usage scenario where a developer is relying on the mapped functions to conduct a migration, providing the results of *FC*, with low precision, leads to potential introduction of many wrong mappings, which hinders its practicality as developers are less likely to trust it. In this situation, higher precision is critical, even at the cost of a lower recall. On the other

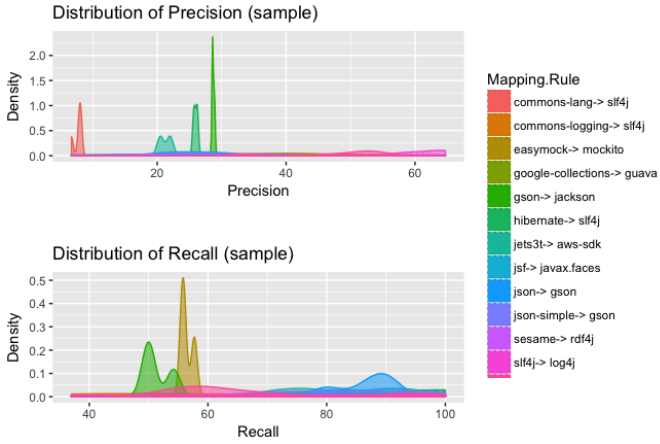

Figure 7: Density of precision and recall of *FC* for all t_{rel} values.

hand, considering a scenario where there is a large set of diverse functions to be replaced, *FC* is more likely to provide candidate replacing function(s), for each retired function, as its high recall allows a better coverage of all previously replaced functions.

To summarize, the high precision of *FS* and the high recall of *FC* naturally drives our future research into designing a hybrid mapping generation algorithm that benefits from combining both approaches to provide the community with a more practical set of mappings.

Results for RQ2. Since the *FC* approach relies on a user-defined filtering threshold t_{rel}, it is important to tune it with respect to each input i.e., migration rule. Thus, for a fair comparison, we measure the performance of *FC* in terms of precision and recall while varying t_{rel} between 0 and 1, for each migration rule. Figure 7 reports the density of precision and recall while varying t_{rel} between 0 and 1.

The *FC* is flexible in allowing a predefined filtering value that aims in reducing the number of false positives. For each migration rule, Figure 8 shows the impact of t_{rel} on the precision and recall of the correct mappings.

Interestingly, the number of mappings per-fragment affects selecting t_{rel} for the best precision and recall. For example, when the number of mappings per-fragment is less than or equal to 2, such as fragments found in *sesame→rdf4j*, *commons-lang→slf4j*, and *jsf→javax.faces* in Table 4, the filtering value t_{rel} has no impact on precision and recall. We can conclude the tuning becomes irrelevant when the number of mappings per-fragment is low. On the other hand, few fragments have more than 1-to-2 mappings, such as *commons-lang→slf4j, and jets3t→aws-sdk*, are affected by the variation of t_{rel}. We experience such impact at its highest in *easymock→mockito*, where every t_{rel} gives a different precision and recall ratio.

Based on the above-mentioned observations, we are in line with the previous study that there is no golden value of t_{rel} for achieving an acceptable trade-off between precision and recall, for all rules combined.

Results for RQ3. To answer the RQ3, we selected 4 migrations with the least number of fragments. The existence of mappings was already validated during the validation process described in subsection 4.3, but in this case, we are searching for any missing fragments: for the removed library, we automatically collect all commits containing the removal of its dependencies and then manually verify whether they were all captured in the fragments. Then we calculate the precision and recall to assess the performance of our mining process. Precision is calculated as the ratio between the number of correctly-included removed functions and the total number of included removed functions. The recall is the measurement of the ratio between the number of correctly-included removed functions and the total number of manually-verified removed functions. We report our findings in terms of precision and recall in Table 6.

We observe in Table 6 that both approaches were successful in avoiding false positives when extracting fragments. Also, they didn't miss any fragment when the number of fragments to extract is relatively low (7 and 18 respectively for jets3t and sesame), except for finding a higher number of fragments. In this case, capturing all fragments is important to increase the overall accuracy of the mappings and to cover as many migrated functions as possible. As shown in Table 6, both approaches extract correct fragments as their precision for the 4 rules is maximal, but the commit-based approach outperformed the version-based approach in detecting a higher number of true positive fragments. This can be explained by the fact that the number of versions is, in general, significantly lower than the number of commits. For example, the Hudson[8] project has 10 versions and 11,332 commits. Using the version-based extraction algorithm a comparison between two version commits is actually searching for fragments in several thousands of merged commits, which adds changes unrelated to the migration. Thus, this makes the detection of fragments difficult when the set of removed and added functions may no longer be co-located in the same block. On the other hand, the commit-based extraction algorithm compares consecutive commits, which facilitate the detection of changes between files, and so, reduces the chance of missing any fragment. Note that, in this study, the ratio of overall *instant migrations* is 37% while the ratio of overall *delayed migrations* is 63%. Table 7 depicts the fragments extraction frequency of both approaches along with the percentage of instant and delayed migration segments for each rule.

According to Table 7, when the percentage of instant migrations is close to the delayed migrations, like in *easymock→mockito* and *json→gson*, where the percentage is respectively 52% and 34.5%, the average number of fragments extracted by the version-based extraction is close to the commit-based frequency. However, for *commons-lang→slf4j* and *slf4j→log4j*, where percentages of instant migrations are 0.7% and 17.8%, respectively, the version-based extraction frequency has drastically decreased compared to the commit-based frequency. This fact is due to the difficulty for the version-based approach of locating fragments when they are distributed across multiple commits. In fact, the percentage of instant and delayed migrations vary from one rule to another since it depends on multiple factors such as the number of projects involved per rule, the length of segments and the number of source files

[8]https://goo.gl/A9Mnui

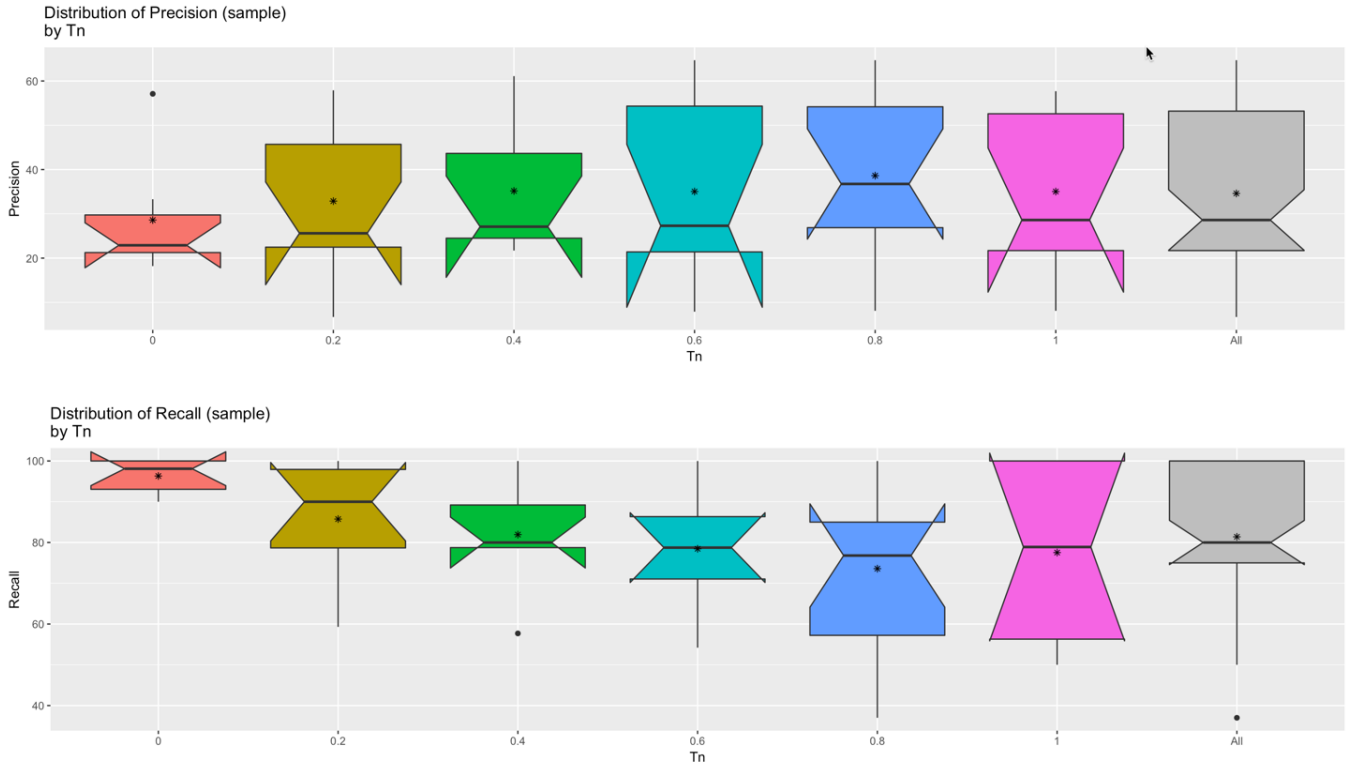

Figure 8: Distribution of Precision and Recall in all projects combined for all t_{rel} values.

Table 6: Precision and recall of the fragment extraction.

Rule (Source →Target)	Version-Based		Commit-Based	
	U-Precision	U-Recall	U-Precision	U-Recall
slf4j→log4j	100%	83%	100%	100%
commons-lang→slf4j	100%	89%	100%	100%
json→gson	100%	100%	100%	100%
easymock→mockito	100%	100%	100%	100%

Table 7: Average number of fragments per migration rule.

Rule (Source →Target)	Average number of fragments		Migration Type	
	Version-based	Commit-based	Instant	Delayed
slf4j→log4j	5.75	9.1	17.8%	83.2%
commons-lang→slf4j	4.7	8.3	0.7%	99.3%
json→gson	5	6.2	34.5%	56.5%
easymock→mockito	34	40	52%	48%

changed in each segment. It is hard to draw conclusions without deeper analysis of the fragment extraction when taking into account these factors.

Overall, we can conclude that the commit-based approach was more successful in localizing fragments, regardless of the ratio of the delayed migration, which is very important for improving the accuracy of the process of generating mappings. This explains how we were successful in increasing the overall performance of state-of-art approaches in comparison with the previous study.

7 THREATS TO VALIDITY

We report the following factors that may negatively influence our replication study. We classify them into construct, internal, and external factors.

Construct Validity. The qualitative analysis was performed by the authors who manually validated every unique mapping then used an automated script to validate all its occurrences in the other fragments. There is a risk of missing fragments which we believe will not significantly impact the outcomes of the algorithms since two of them rely on multiple occurrences of similar mappings, and missing a few will not propagate through all migration rules. Also, the third algorithm does not depend on the frequency of occurrences. We also may have mis-evaluated some mappings. This will uniformly impact the 3 approaches, but, will not favor one over the others. To mitigate this, we made sure to analyze commits from different projects, for the same migration rule, to be sure of the mapping.

Internal Validity. One of the most critical threats to our work is the utilization of the *Diff* function to approximate the changed code blocks. We implemented all the data collection and the approaches under comparison from scratch. We believe that the function is well-recognized and used often in the industry. Furthermore, it is the built-in function for existing operating systems and most importantly for GitHub, the host of all the projects we parsed. Using this function drastically facilitates our mining process. Moreover, there may be errors in our tool that may engender missing some commits. We carefully paid attention when implementing the mapping generation algorithms. We verified our findings with the original findings of the previous paper as one way to ensure correctness. We considered the developer's decisions, which we mine, as the ground truth for generating ideal function mappings. Thus, our approach is subject to developers errors. We believe that this may not be problematic since we are mining a large number of projects developed by a diverse set of developers.

When mining open source projects, we rely on the *Project Object Model* file for the identification of used libraries, which limits our studies to projects using *Maven* [9]. Yet, we were able to shortlist a large number of projects for the experiments. Also, the detection of a project's hidden dependencies is out of our scope.

Another important threat is the choice of migration rules, we have relied on existing studies to extend the set of migrations between libraries, and some of these migrations may not be necessarily correct, for example, *jsf-api⟶javax.faces* has been detected in the previous studies as a migration rule but as we perform the manual validation, we identified them as two different implementations of the same API library. Our mining algorithm identified segments for this migration as the two instances of the same API have different identifiers.

External Validity. Our study was limited to one-to-one migration rules, which does not reflect the reality of the existence of one-to-many library migrations. This represents a threat to the generalization of our approach. We plan in the future to take into account any library migration regardless of its cardinality. Another threat concerns the chosen projects, which are an exclusively open source. Thus, our results cannot be generalized for all types of projects.

[9]https://maven.apache.org/what-is-maven.html

8 RELATED WORK

Several studies focused on understanding how developers perceive API related function changes. In the context of library updates, many studies have been proposed to capture the needed changes on the client source code applied along with API migration [9, 11, 18, 19]. They used textual similarity between structures and function signatures as a means to identify identical functions between multiple library versions. This approach can be adapted in the context of library migration if applied on code fragments. This drove us to consider it as a potential approach to map similar libraries [11]. Another study relevant to our work has been conducted by Schäfer et al. [14]. They analyze changes in function call locations to extract the fragments of added/removed functions. They compute the associated rules from fragments before filtering them using the similarity of function signatures. Their approach allocates one function to each function call. This favors the 1-1 function mapping and hinders the existence of other added (resp. removed) functions in case of N-M function mapping i.e., replacing one or many functions with one of many functions within the same fragment. Teyton et al. [16] extended this to support all possible cardinalities of functions mappings. For a given input migration rule, they extracted all fragments, then applied the Cartesian Product between the two sets of removed and added functions. This generates all the possible combinations of mappings that may have occurred between the set of source and target library functions. The frequency of identical combinations is calculated throughout all the studied projects. Finally, an acceptance threshold is set so that any combination whose frequency is higher than the threshold is considered a correct function mapping. In contrast with previous studies, this approach was similarity agnostic since it is robust to libraries variations in design, naming conventions and vocabulary. On the other hand, it exclusively relies on existing migrations between two given libraries to provide mappings. Lastly, its performance, in terms of accuracy, depends on the frequency of such migrations across projects, as demonstrated later in the experiments.

A dynamic analysis was also used by Gokhale et al. [3] to infer likely mappings between the APIs of Java2 Mobile Edition and Android graphics. Their approach was specific to the given libraries. Kabinna et al. [7] mined the migration of 9 logging libraries in Apache software foundation projects. Their findings show that the majority of the 49 detected migrations were successful, but the process is error-prone with an average of two post-migration bugs even when experienced developers were accomplishing the migration task.

In this paper, we conduct a comparative study between Teyton et al. [16] and Schäfer et al. [14] by including one study that is representing the set of similarity-based approaches. We adopted the approach of Nguyen et al. [11] to the context of detecting mappings. In the next section, we provide the challenges of detecting mappings at the function level along with detailing how each approach generates the mappings using an illustrative example.

9 CONCLUSION

This paper enhanced an existing study for addressing the problem of mining developer decisions in migrating third-party libraries. We included a larger set of migration rules and a wider set of open

source projects. Furthermore, we extended the fragment extraction process to generate a larger set of potential candidate mappings between functions which provide more coverage for migrations between APIs. Additionally, we challenged the previous study's approach by adapting one state-of-art function similarity approach, namely *FS*.

Our main findings include increasing the range of function mappings, and so providing the community with a richer set of changes between libraries. Our results show that the performance of the compared approaches depends essentially on the density of fragments and the number of unique removed/added functions per fragments. These factors drastically change the performance of the 3 algorithms. As for tuning the algorithm of the previous study, there is no silver bullet that guarantees its performance for a given migration rule and a set of input projects. In addition, our proposed commit-based extraction generates a higher, yet correct, number of fragments, which improved the performance of the previous algorithm compared to the original paper as well as the performance of the competitive approaches.

In our future work, we plan on mainly merging the three approaches by combining their three measurements into one weighted fitness. We also plan on tackling the one to many library migrations and address its related challenges. We plan on extending our current analysis to better understand how the density of fragments impacts the performance of the existing algorithms. Also, we plan on extending the analysis conducted in RQ3, by including all the migration rules, and analyzing the performance of commit-based and version-based approaches while tracing other factors such as the number of changes files per segment.

Since the application of FC requires the tuning of the t_{rel}, the lack of golden value may negatively impact on the overall matching performance and could be a barrier for adopting FS in practical scenarios. Therefore, we plan on automating the finding of an acceptable threshold value.

As previously indicated, commit-level mining is expensive, we plan on re-designing our tool to run on multiple instances, by parallelizing the comparison of commits pairs. This will drastically decrease the time needed to mine all projects for a given migration rule. Finally, we will also explore the possibility of building a library recommendation system at the function level that learns from developers previous decisions to recommend similar library functions in similar contexts.

REFERENCES

[1] H. Alrubaye, M. W. Mkaouer, and A. Peruma. Variability in library evolution: An exploratory study in java libraries. In B. M. Ivan MistrÃŋk, Matthias Galster, editor, *Software Engineering for Variability Intensive Systems: Foundations and Applications*, chapter 13. Taylor & Francis Group, LLC/CRC Press, 2018.

[2] H. P. Enterprise. Hpe security research: Cyber risk report 2016. *Hewlett Packard Enterprise Security Research, Bracknell, UK*, 2016.

[3] A. Gokhale, V. Ganapathy, and Y. Padmanaban. Inferring likely mappings between apis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 82–91. IEEE Press, 2013.

[4] A. Hora, A. Etien, N. Anquetil, S. Ducasse, and M. T. Valente. Apievolutionminer: Keeping api evolution under control. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 420–424. IEEE, 2014.

[5] A. Hora and M. T. Valente. apiwave: Keeping track of api popularity and migration. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 321–323. IEEE, 2015.

[6] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

[7] S. Kabinna, C.-P. Bezemer, W. Shang, and A. E. Hassan. Logging library migrations: a case study for the apache software foundation projects. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 154–164. IEEE, 2016.

[8] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE*, volume 7, pages 333–343. Citeseer, 2007.

[9] S. Kim, K. Pan, and E. J. Whitehead. When functions change their names: Automatic detection of origin relationships. In *In Reverse Engineering, 12th Working Conference on*, pages 10–pp. IEEE, 2005.

[10] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 117–128. IEEE, 2002.

[11] 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 *ACM Sigplan Notices*, volume 45, pages 302–321. ACM, 2010.

[12] A. Ouni, R. G. Kula, M. Kessentini, T. Ishio, D. M. German, and K. Inoue. Search-based software library recommendation using multi-objective optimization. *Information and Software Technology*, 83:55–75, 2017.

[13] R. Pandita, R. P. Jetley, S. D. Sudarsan, and L. Williams. Discovering likely mappings between apis using text mining. In *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, pages 231–240. IEEE, 2015.

[14] T. Schäfer, J. Jonas, and M. Mezini. Mining framework usage changes from instantiation code. In *Proceedings of the 30th international conference on Software engineering*, pages 471–480. ACM, 2008.

[15] C. Teyton, J.-R. Falleri, and X. Blanc. Mining library migration graphs. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 289–298. IEEE, 2012.

[16] C. Teyton, J.-R. Falleri, and X. Blanc. Automatic discovery of function mappings between similar libraries. In *In Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 192–201. IEEE, 2013.

[17] C. Teyton, J.-R. Falleri, M. Palyart, and X. Blanc. A study of library migrations in java. *Journal of Software: Evolution and Process*, 26(11):1030–1052, 2014.

[18] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim. Aura: a hybrid approach to identify framework evolution. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 325–334. IEEE, 2010.

[19] Z. Xing and E. Stroulia. Api-evolution support with diff-catchup. *IEEE Transactions on Software Engineering*, 33(12):818–836, 2007.