Mining Scala Framework Extensions for Recommendation Patterns

Yunior Pacheco[1,2], Jonas De Bleser[1], Tim Molderez[1], Dario Di Nucci[1], Wolfgang De Meuter[1], Coen De Roover[1]

[1]*Vrije Universiteit Brussel*
Brussels, Belgium
{ypacheco, jdeblese, tmoldere, ddinucci, wdmeuter, cderoove}@vub.be

[2]*Pinar del Rio University*
Pinar del Rio, Cuba

***Abstract*—To use a framework, developers often need to hook into and customise some of its functionality. For example, a common way of customising a framework is to subclass a framework type and to override some of its methods. Recently, Asaduzzaman *et al.* defined these customisations as *extension points* and proposed a new approach to mine large amounts of Java code examples and recommend the most frequently used example, so called *extension patterns*. Indeed, recommending *extension patterns* that frequently occur at such *extension points* can help developers to adopt a new framework correctly and to fully exploit it.**

In this paper, we present a differentiated replication study of the work by Asaduzzaman *et al.* on Java frameworks. Our aim is to replicate the work in order to analyse extension points and extension patterns in the context of Scala frameworks. To this aim, we propose SCALA-XP-MINER, a tool for mining *extension patterns* in Scala software systems to empirically investigate our hypotheses.

Our results show that the approach proposed by the reference work is also able to mine extension patterns for Scala frameworks and that our tool is able to achieve similar *Precision*, *Recall* and *F-measure* compared to FEMIR. Despite this, the distribution of the extension points by category is different and most of the patterns are rather simple. Thus, the challenge of recommending more complex patterns to Scala developers is still an open problem.

***Index Terms*—Framework, Extension Points, Usage Patterns, Graph Mining, Scala Language, Mining Software Repository**

I. INTRODUCTION

A significant part of software development involves becoming familiar with APIs from different libraries and frameworks. Libraries and frameworks enable code reuse, provide high-level abstractions for common tasks, and help unify the programming experience [1]. However, the flexibility of large libraries and frameworks usually comes at the expense of sophisticated APIs that must be accessed and combined. Additionally, these usage patterns require specialised knowledge about the behaviour of the API [2] to be used correctly. Thus, making efficient use and exploiting all the possibilities offered by libraries and frameworks can be quite difficult due to specific requirements and relations between its components. In general, the larger and more sophisticated the library or framework is, the harder this challenge. Furthermore, the library or framework may not be documented completely or clearly [1].

When using a framework, it is either necessary or common to extend its functionality. Several studies analysed how developers use libraries in software systems; providing tools to explore and navigate usage examples [3]–[5], documenting techniques [6], and recommending usage patterns obtained from mining code examples [7], [8]. However, there are relatively few studies that focus on common ways to *extend* a framework and to provide recommendations to developers in this respect.

In this paper, we replicate the work of Asaduzzaman *et al.* [9] for mining framework extension patterns in Java code in the context of Scala frameworks. These patterns can be derived by grouping frequently occurring framework method calls that provide developers a way to extend the framework (*e.g.,* methods that expect a subclass of a framework class) and can provide useful suggestions to developers about how to extend a framework. To this end, given a framework, a significant number of projects that are using it need to be mined in order to get usable and useful recommendations. Manually inspecting every framework extension is not feasible, thus we present SCALA-XP-MINER, a tool for mining extension patterns in Scala code.

SCALA-XP-MINER is needed not only because Scala is gaining traction in industry, but also because of the unique features of this programming language that may impact the way in which framework extension points and patterns occur. For example, features such as case classes, implicit type conversions, companion objects, implicit classes, implicit parameters, by-name parameters, *etc.* might introduce new possibilities to extend frameworks. Despite this, first we would like to analyse to what extent the approach of Asaduzzaman *et al.* [9] is able to mine extension patterns in Scala code. Therefore we conducted a differentiated replication study [10] with the aim of analysing the diffusion and the characteristics of the extension points in Scala code. The replication is differentiated as we re-implemented the original approach [9] in a different context (*i.e.,* Scala software systems). In particular, we analysed 467 open-source Scala projects to find extension patterns for five popular Scala frameworks: SPARK, HADOOP, PLAY, MOCKITO, and AKKA.

Our results show that the approach proposed by Asaduzzaman *et al.* [9] is able to mine extension patterns for Scala code. Furthermore, our tool achieves a similar *Precision*, *Recall* and *F-measure* as the work of Asaduzzaman *et al.* [9]. Despite this, the distribution of the extension points by category is different. In particular, we observe that few patterns belong to the *Extend* category and none of the extension patterns belong to the *Implement* category.

In summary, the contributions of this work are three-fold:

- a differentiated replication study [10] of the work by Asaduzzaman *et al.* [9] in the context of Scala code.
- SCALA-XP-MINER, a tool that recommends patterns for the extension points of a given framework by mining for patterns among the extension point usages in Scala projects that use the framework
- a comprehensive replication package [11] including all the raw data and the scripts used in the study.

Structure of the paper. The remainder of this paper is organised as follows. Section II describes FEMIR, the tool [9] developed by Asaduzzaman *et al.* for mining framework extension patterns in Java software systems, while Section III presents SCALA-XP-MINER, a tool that replicates the approach implemented in FEMIR to mine extension patterns in Scala software systems. Section IV discusses the replication study that we conducted on a dataset of Scala projects. The threats to validity are discussed in Section V, while Section VI presents an overview of existing techniques that assist developers in using frameworks, and Section VII concludes the paper.

II. FEMIR: RECOMMENDING EXTENSION EXAMPLES IN JAVA

In this section, we provide an overview of FEMIR, together with a summary of its empirical evaluation.

A. Mining Extension Patterns using FEMIR

Extension Points have been defined by Asaduzzaman *et al.* as *"means provided by a framework, that allow developers to customise its behaviour, to meet application specific requirements"*. For example, frameworks can define an extension point as a public method that takes a framework type as one of its parameters. A common way of using such an extension point (*i.e.,* an *extension point usage*) is by calling the method with an instance of the framework type itself, or with an instance of a user-defined subtype that overrides some of the inherited methods [9].

```
class SparkContext(config: SparkConf) {
  def addSparkListener(listener: SparkListenerInterface) = ...
}
class StageInfoRecorderListener extends SparkListener {
    override def onJobStart(jobStart: SparkListenerJobStart): Unit = ...
    override def onStageCompleted(stageCompleted: SparkListenerStageCompleted):
       Unit = ...
}
case class StageMetrics(sparkSession: SparkSession) {
    sparkSession.sparkContext.addSparkListener(new StageInfoRecorderListener)
    ...
}
```

Listing 1. Example of extension usage of the SPARK framework in Scala.

Listing 1 depicts a simplified example of an extension point and its usage in SPARK. In this example, `addSparkListener`, as well as `onJobStart()`, and `onStageCompleted()` are extension points. As described before, these methods are extension points because they take at least one parameter of a framework type and are defined in a framework class (*i.e.,* `SparkContext`) or in a subclass (*i.e.,* `StageInfoRecorderListener`), respectively. The method call `addSparkListener` on the instance `sparkSession.sparkContext` represents an extension point usage. In this case, the method takes an instance of `StageInfoRecorderListener` which extends from `SparkListener`, which in turn extends from `SparkListenerInterface`.

It is worth noting that the methods `onJobStart()` and `onStageCompleted()` are overridden in the class `StageInfoRecorderListener`. In this way, an instance of this class can modify the default behaviour inherited from the framework.

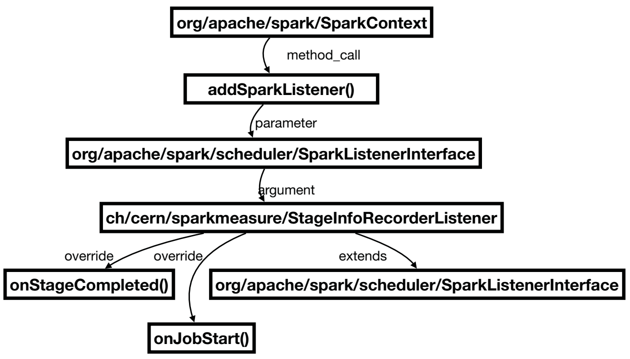
org/apache/spark/SparkContext
method_call
addSparkListener()
parameter
org/apache/spark/scheduler/SparkListenerInterface
argument
ch/cern/sparkmeasure/StageInfoRecorderListener
override
override
extends
onStageCompleted()
org/apache/spark/scheduler/SparkListenerInterface
onJobStart()

Fig. 1. The EXTENSION GRAPH that represents the extension point usage of `SparkContext#addSparkListener` shown in Listing 1.

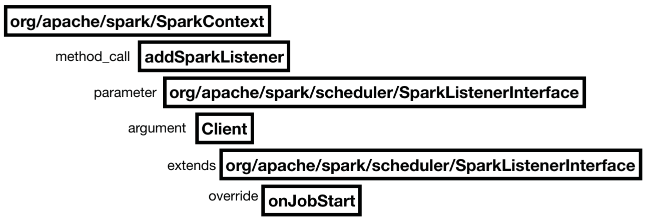
org/apache/spark/SparkContext
method_call
addSparkListener
parameter
org/apache/spark/scheduler/SparkListenerInterface
argument
Client
extends
org/apache/spark/scheduler/SparkListenerInterface
override
onJobStart

Fig. 2. An EXTENSION PATTERN extracted from Figure 1.

Each extension point is represented by means of an *extension graph*. Such a graph consists of several types of nodes: RECEIVER TYPE, METHOD CALL, PARAMETER TYPE, ARGUMENT TYPE, OTHER METHOD CALLS, EXTENDED CLASS, IMPLEMENTED INTERFACE, OVERRIDING METHOD, and FRAMEWORK METHOD CALL. These extension graphs are built by parsing and analysing the source code of a project that uses the framework for which we want to obtain the extension points. Figure 1 shows the extension graph generated for the example shown in Listing 1.

The subgraphs that most frequently occur in the extension graphs are called EXTENSION PATTERNS. These patterns are useful to describe how an extension point is commonly used.

The miner uses an iterative and incremental approach to mine these extension patterns. First, a one-node subgraph is generated for each input extension graph. These one-node graphs are compared each other and the most frequent ones are grown by adding an adjacent node from the corresponding input extension graph. The growing process is iterated until all nodes of the input extension graphs have been considered. To further improve the quality of the extension patterns, the support of the nodes that were not previously included is computed. The support of a node is defined as the number of extension graphs that contain the node, divided over the number of all extension graphs. A node is added if this measure is higher than a threshold δ, for which the default value is 0.3.

It is clear that there are multiple kinds of extension patterns according to how the framework is to be extended. Asaduzzamann *et al.* [9] grouped these patterns and created a taxonomy of four *categories* to describe the complexity of the pattern:

(i) SIMPLE: an instance of a framework class is passed as an argument to the extension point without modifying it;
(ii) CUSTOMISE: before passing the argument of a framework type to the extension point, a number of state changing methods are called on it;
(iii) EXTEND: the argument to the extension point is an instance of a new class that extends a framework class;
(iv) IMPLEMENT: the argument to the extension point is an instance of a new class that implements a framework class.

Extension patterns are very useful to aid the developer in using a framework. For example, the extension pattern for the `addSparkListener` is shown in Figure 2. This pattern belongs to the category EXTEND and shows that a possible use of the extension point `addSparkListener` is to provide a subclass which extends `SparkListenerInterface` and overrides `onJobStart`.

B. *Recommending Extension Patterns in* FEMIR

FEMIR returns a subset of extension patterns given a framework type. In particular, it considers all extension point usages, but only returns those of which the receiver type corresponds to the given framework type. These patterns are then sorted by their category and frequency in the mined dataset. Finally, based on the selected *recommendation strategy* the top-n patterns are shown to the developer. These strategies are:

- LOCAL: it recommends the top-n patterns within the same category.
- GLOBAL: it recommends the top-n patterns regardless of the category.
- D: after applying the local strategy, it retrieves all the extension graphs that contain the patterns and recommends only the patterns of which their extension graph contains the highest number of different kinds of nodes. We enumerated the different kinds of nodes in Section II.

After having selected a recommended pattern, the related code example is displayed. In the following subsection we detail the process adopted for evaluating FEMIR.

C. *Evaluation of the Patterns Recommended by* FEMIR

The authors analysed how developers extend five widely-used open-source frameworks: SWING, JFACE, JUNIT, JUNG, and JGRAPHT. For each framework they analysed between 167 to 300 projects.

Given an extension point, the authors first evaluated the effectiveness of FEMIR in recommending an extension pattern that matches the actual usage of the extension point. To do this, they developed an evaluation system that collects the extension graphs from the subject classes for each framework. The evaluation system applies the 10-fold cross validation technique to measure the performance. In particular, it divides the extension graphs in ten folds, each containing the same number of graphs. At each iteration of the evaluation, the patterns are mined in nine folds and then used on the remaining one. Note that for each extension pattern from the test set, FEMIR proposes the top-n extension graphs to recommend. Each recommendation is represented as a graph. The authors computed *precision*, *recall*, and *F-Measure* as follows. Let S and O be two graphs representing the suggested extension graph and the original one respectively.

$$Precision = \frac{|O| \cap |S|}{|S|} \quad Recall = \frac{|O| \cap |S|}{|O|}$$

$$F-Measure = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Note that not all the recommendations were evaluated, but only the top-n (*i.e.,* top-1, top-3, top-5). Given the top-n recommendations, only the extension pattern achieving the best *F-Measure* was evaluated.

Beside evaluating the effectiveness of FEMIR, the authors also evaluated its quality by showing some examples of the proposed extension patterns. Afterwards, they analysed the distribution of the extension patterns with respect to their category and how many patterns were found per framework type. The latter analysis was needed to understand whether the suggestions provided by FEMIR are useful for developers. Finally, they studied to what extent extension points are used together by grouping the extension graphs with respect to their receiver type. In a following discussion, they clarified aspects such as the effectiveness of FEMIR in detecting different categories of extension patterns, the quality of a canonical form representation, and the effect of a threshold value δ on the effectiveness. Lastly, they analysed the efficiency of the tool in terms of runtime performance. We discuss the results in section IV.

In this paper, we conduct a differentiated replication study of the work of Asaduzzaman *et al.* [9] in the context of Scala software systems. In summary, the approach of Asaduzzamann *et al.* [9] analyses source code and generates extension graphs for each extension point. These graphs are processed using a frequent subgraph mining algorithm to extract the extension patterns. These patterns are then grouped in categories according to a taxonomy and recommended to the developers. To this aim, we propose SCALA-XP-MINER, a tool that mines

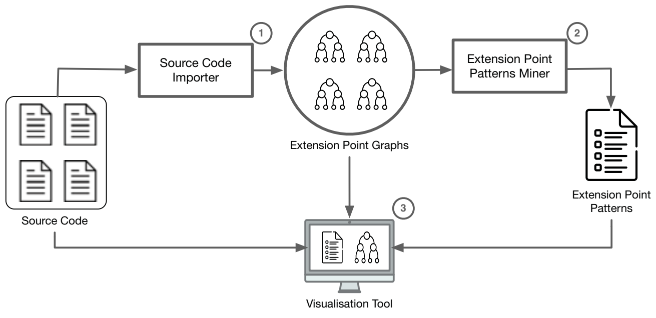

Fig. 3. Overview of the approach

extension patterns in Scala software systems. The next section presents a complete overview of SCALA-XP-MINER.

III. SCALA-XP-MINER: Recommending Extension Examples in Scala

Our framework for mining extension patterns in Scala projects consists of three components: a source code importer, a pattern miner, and a visualisation tool. Figure 3 depicts the interactions between these components.

We follow a 3-step approach similar to the one proposed in the reference work [9]. First, we build a graph for each of the extension usages in the Scala projects that depend on the framework under analysis. Next, we mine extension patterns using the information previously extracted. Finally, we visualise the input and output of the mining algorithm in a way that allows the developer to browse and understand the results.

1. Source Code Importer. The importer takes the source code of framework clients as input and collects information on framework usages: for each framework method call in the source code, the importer statically checks whether it corresponds to an extension point. To be considered, the method call must have at least one parameter that is related to a framework type. For each extension point, we collect the method name, the return type, and the types of the parameters.

To construct the extension graphs, the importer resolves the types of the receiver and the arguments of the extension point. Thus, the type hierarchy and the list of overridden methods must be computed. These steps are necessary but not sufficient to completely support Scala. For example, in Java there is a difference between `implements` and `extends`, while in Scala this concept does not exist. In this case, we consider that a class `implements` an interface only when its first parent is a trait; while in all the other cases it `extends` a class. In general, SCALA-XP-MINER has to support Scala-specific features such as singleton and companion objects, implicits, lambda expressions, *etc.* to analyse Scala source code correctly. It also identifies method calls of which the receiver is either the same as the receiver of the extension point or refers to one of its arguments. This is needed because more complex patterns could be composed of multiple method calls. The importer extracts the required syntactic and semantic information through the SCALA-META[1] library.

2. Extension Patterns Miner. The miner is responsible for mining the extension patterns. The frequent subgraph mining algorithm, used in the miner, takes as input the set of extension graphs generated by the importer in the previous step. The frequent subgraph mining algorithm is a variant of the Apriori algorithm [12]. This variant, unlike the Apriori algorithm, does not use a support parameter to determine the set of subgraphs to be expanded, but instead, takes the top-k frequent candidate subgraphs. To obtain extension patterns for a given project, we mine projects and represent the found extension points by means of extension graphs. Finally, we use the mining algorithm to obtain the extension patterns such as the pattern shown in Figure 2.

3. Visualization Component. The visualisation component is used to configure the tool and to browse through and inspect its results. The user can select the Scala projects to import through the SOURCE CODE IMPORTER. The EXTENSION PATTERNS MINER then analyses the projects to discover extension usages of a specific framework (given as input). After the computation, the tool displays a list view and a graph view of the extension graphs built by the importer and that constitute the input to the mining algorithm. Finally, the tool supports inspecting the frequent extension patterns uncovered by the mining algorithm, and comparing them to their matches among the input data, to corroborate the validity of the mining process.

IV. Empirical Study Design and Results

The *goal* of our empirical study is to replicate the work of Asaduzzaman *et al.* [9], with the *purpose* of understanding the distribution of extension points and the possibilities for recommending extension patterns for Scala frameworks. This is from the perspective of both researchers and developers: the former are interested in understanding to what extent extension points are used throughout Scala projects, while the latter are interested in using extension patterns. To this end, our study is designed to answer, in the context of Scala, the same research questions as those from the reference work:

- **RQ$_1$:** *To what extent are the proposed recommendations accurate?*

[1]https://github.com/scalameta/scalameta

TABLE I
Characteristics of the Projects Considered in the Replication Study for each Framework

Framework	# Projects	# Files	# Classes	# Methods	# LOC
Spark	102	13,144	19,448	86,156	2,148,572
Akka	107	10,836	18,615	63,865	1,158,327
Mockito	99	10,097	12,845	43,159	992,510
Hadoop	58	10,344	15,324	71,447	1,799,918
Play	101	7,898	10,642	40,656	741,414

TABLE II

ACCURACY OF THE EXTENSION PATTERNS SUGGESTED BY SCALA-XP-MINER FOR EACH FRAMEWORK AND STRATEGY ALONG WITH THE STATISTICS ON THE EXTENSION GRAPH COMPOSITION (*RMC*: RECEIVER METHOD CALL, *AMC*: ARGUMENT METHOD CALL, *E*: CLASS EXTENSION, *I*: CLASS IMPLEMENTATION, *FMC*: FRAMEWORK METHOD CALL, *O*: OVERRIDDEN METHOD, *Other*: OTHER NODE TYPES). *ARS* IS THE AVERAGE RECOMMENDATION SIZE, WHILE *No REC* REPRESENTS THE PERCENTAGE OF RECEIVER OBJECTS FOR WHICH IT WAS NOT POSSIBLE TO PROVIDE SUGGESTIONS

Framework	**Strategy**	**RMC AMC**	**E I**	**O FMC**	**Other**	**ARS**	**No REC**	**Precision**			**Recall**			**F-Measure**		
								Top-1	**Top-3**	**Top-5**	**Top-1**	**Top-3**	**Top-5**	**Top-1**	**Top-3**	**Top-5**
Spark	Global	9.50%	0.03%	0.47%	90.00%	4	50.83%	0.92	0.95	0.96	0.85	0.89	0.91	0.88	0.91	0.93
	Local					4	53.03%	0.93	0.95	0.97	0.85	0.89	0.91	0.88	0.92	0.93
	D					4	53.18%	0.90	0.94	0.96	0.84	0.88	0.91	0.86	0.91	0.93
Akka	Global	3.96%	0.17%	1.42%	94.45%	3	27.46%	0.96	0.97	0.97	0.88	0.91	0.92	0.91	0.94	0.94
	Local					3	28.93%	0.96	0.97	0.98	0.88	0.92	0.93	0.91	0.94	0.95
	D					3	28.78%	0.95	0.97	0.97	0.88	0.91	0.92	0.91	0.94	0.94
Mockito	Global	11.55%	0.03%	0.03%	88.39%	3	27.31%	0.96	0.97	0.97	0.88	0.91	0.92	0.91	0.94	0.94
	Local					3	0.65%	0.96	0.97	0.97	0.72	0.79	0.82	0.82	0.87	0.88
	D					3	0.65%	0.96	0.97	0.97	0.72	0.77	0.80	0.82	0.85	0.87
Hadoop	Global	3.70%	0.06%	0.03%	96.21%	4	15.06%	0.96	0.96	0.96	0.97	0.98	0.98	0.96	0.97	0.97
	Local					4	23.49%	0.98	0.98	0.98	0.97	0.98	0.98	0.98	0.98	0.98
	D					4	23.64%	0.98	0.98	0.98	0.97	0.98	0.98	0.98	0.98	0.98
Play	Global	17.02%	0.10%	0.06%	82.82%	4	11.40%	0.92	0.94	0.96	0.85	0.88	0.90	0.87	0.90	0.92
	Local					4	15.36%	0.93	0.95	0.97	0.94	0.88	0.90	0.87	0.90	0.92
	D					4	15.38%	0.92	0.95	0.96	0.85	0.88	0.90	0.88	0.90	0.92

- **RQ$_2$:** *To what extent is the approach able to provide accurate suggestions for each category of extension patterns?*
- **RQ$_3$:** *How are extension patterns distributed by category?*
- **RQ$_4$:** *How are extension points distributed across framework classes?*
- **RQ$_5$:** *To what extent do developers use multiple extension points to extend a component?*
- **RQ$_6$:** *What is the impact of algorithm tuning on the performance?*
- **RQ$_7$:** *How does the approach perform in terms of execution time?*

The goal of **RQ$_1$** is to determine the accuracy of SCALA-XP-MINER in recommending extension patterns, while **RQ$_2$** focuses on the ability of the approach to suggest useful recommendations for each category of extension points. With **RQ$_3$**, we assess the distribution of extension patterns in terms of the categories defined in the reference work. In **RQ$_4$** we examine the usefulness of the approach in terms of the number of suggestions proposed for a given receiver type. With **RQ$_5$** we analyse to what extent developers use multiple extension points when they need to extend a component of a framework. **RQ$_6$** analyses the effect of tuning the parameter δ used in FEMIR-GLOBAL and FEMIR-LOCAL. Finally, **RQ$_7$** investigates the performance of the approach in terms of the execution time needed for training the algorithm and showing the recommendations.

In contrast to the reference work, we focus on frameworks that provide a Scala API and on software systems implemented in Scala and hosted on GITHUB. We considered only repositories that adhere to the requirements of using SBT 0.13+ and Scala 2.11+ because we rely on SCALA-META to generate semantic information[2]. To select the final subset, we carried out a preliminary study to find the most prominent Scala frameworks. We grouped the packages imported by 177.828 Scala software systems, and concluded that SPARK, AKKA, HADOOP, MOCKITO, and PLAY are the most used frameworks. These five frameworks provide APIs both for Scala and Java, and these can be used interchangeably. Therefore, we could not disambiguate between them and collected all framework usages from both APIs. Given these frameworks, we analysed the top-250 projects in terms of their number of stars, a well-known metric of popularity [13]. In particular, first we mined Scala repositories that imported at least one class from the considered frameworks (*e.g.,* `org.apache.spark.*`). Then, we ranked them by stars and analysed the top-250 projects for each framework. Note that we were not able to build all projects and therefore had to discard some. We used the number of stars as a proxy for project size. The characteristics of the successfully analysed repositories are shown in Table I. We will now discuss the design and results for each research question in detail:

[2]SBT 0.13 was released in 2013; Scala 2.11 was released in 2014.

A. RQ$_1$ — To what extent are the proposed recommendations accurate?

Design. Like the reference work, our tool produces its recommendations for a given extension point using one of three strategies: LOCAL, GLOBAL, and D (see Section II-B). To evaluate the accuracy of these recommendations, we apply the 10-fold cross validation technique. Given every recommended extension pattern from the training set and an actual usage from the test set, the tool computes *precision*, *recall*, and *F-Measure* as previously defined in section II-B. Between the top-n recommendations, the extension pattern achieving the best *F-Measure* is evaluated. In other words, for each input extension graph from the test set, the top-n patterns from the training set able to suggest it are computed. Please note that in case SCALA-XP-MINER is not able to return all the requested recommendations, we consider only the available ones. Among these patterns, the one achieving the best *F-Measure* is selected

TABLE III
ACCURACY OF THE TOP-N RECOMMENDATIONS FOR EACH EXTENSION PATTERN CATEGORY ON SPARK

Extension Pattern Category	Precision			Recall			F-Measure		
	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
Simple	0.93	0.95	0.97	0.87	0.91	0.93	0.89	0.93	0.95
Customise	0.94	0.95	0.96	0.75	0.78	0.78	0.81	0.84	0.84
Extend	-	-	-	-	-	-	-	-	-
Implement	-	-	-	-	-	-	-	-	-

and compared to the actual extension usage. The results are then aggregated using the *mean* operator.

FEMIR Results. The F-Measure obtained by FEMIR-GLOBAL ranges between 67% and 82% for the top-5 recommendations. FEMIR-LOCAL performance is close to FEMIR-GLOBAL. In particular, FEMIR-LOCAL improves the precision at the cost of recall. Finally, FEMIR-D performs worse than FEMIR-GLOBAL in all the cases but one (*i.e.,* JGRAPHT).

SCALA-XP-MINER Results. Table II shows the *precision*, *recall*, and *F-measure* of the extension patterns recommended by SCALA-XP-MINER. In summary, we notice that in the context of Scala frameworks, the three strategies do not have a significant impact on the results. Using the GLOBAL strategy, precision ranges between 92% and 97%, and recall between 85% and 98%. As reported also in the reference study, the LOCAL strategy results in improved precision (between 93% and 98%) but a slightly worse recall (between 72% and 98%). We also confirm that the D strategy does not guarantee better results. In general, we observe that our precision and recall achieved on Scala frameworks are better than those achieved on Java frameworks by the reference work [9]. We believe that this is due to the nature of the extension patterns found. Looking at the composition of the extension graphs, we notice that in most of the cases these graphs are composed only by nodes of type *Other* (*i.e.,* RECEIVER TYPE, METHOD CALL, PARAMETER TYPE, ARGUMENT TYPE, as defined in Section II-A).

In terms of size, the recommendations provided by SCALA-XP-MINER consist of three or four nodes on average. This means that most of the Scala framework usages recognized by SCALA-XP-MINER are simpler compared to the Java ones. We also analyse the percentage of cases for which SCALA-XP-MINER is not able to provide a recommendation. It is surprising to find that in case of SPARK for half (*i.e.,* between 50.83 - 43.19%) of the extension points, it is not possible to provide a recommendation; while for the other frameworks this value ranges between 0.54% and 28.78%. Further investigations into the reasons for this behaviour of the approach on Scala frameworks are part of our future agenda.

B. RQ$_2$ — To what extent is the approach able to provide accurate suggestions for each category of extension patterns?

Design. To answer this research question, we repeat the evaluation performed for **RQ$_1$** with respect to the accuracy of the approach in detecting the different types of extension patterns (*i.e.,* SIMPLE, CUSTOMISE, EXTEND, IMPLEMENT). For brevity's sake, we only report the results for SPARK, the most used framework in our dataset. The results for the other frameworks are available in our online appendix [11].

FEMIR Results. FEMIR is able to recommend extension patterns for the four different categories. The patterns belonging to the EXTEND category are detected with higher accuracy.

SCALA-XP-MINER Results. Table III shows the *precision*, *recall*, and *F-Measure* for each extension pattern category. On SPARK, SCALA-XP-MINER is able to recommend only extension patterns belonging to the SIMPLE and CUSTOMISE categories. Nevertheless, we can observe that the extension patterns belonging to the latter category have a lower *recall* compared to the former one, while their *precision* is similar. This behaviour differs from what has been observed in the reference work [9]. For this reason, we analyse a subset of patterns. We achieved similar results when looking at the other frameworks. In general, we find that the higher the size of the actual usage graphs, the lower the *recall* achieved by SCALA-XP-MINER.

C. RQ$_3$ — How are extension patterns distributed by category?

Design. We evaluate the distribution of the extension patterns by category for all the frameworks considered in our replication. It is worth recalling that, when the extension usage graphs are mined to extract the extension patterns, SCALA-XP-MINER assigns to each extension pattern a category, as defined in the taxonomy from the reference paper.

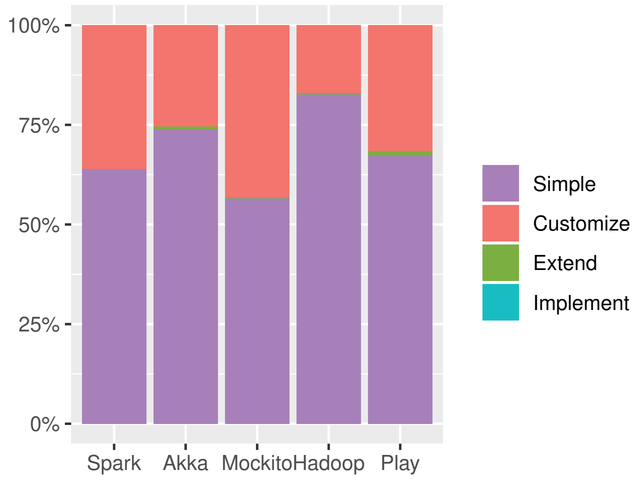

Fig. 4. Distribution of extension patterns by categories for each framework

FEMIR Results. The extension pattern belonging to the CUSTOMISE category are the most commonly occurring category, followed by the category SIMPLE. The extension patterns belonging to the EXTEND and IMPLEMENT categories are not very diffused, but they are the hardest to learn as well.

SCALA-XP-MINER Results. Figure 4 depicts the distribution of extension patterns by category. The most prominent extension category is SIMPLE, which indicates that most of the arguments to an extension point are instances of an actual framework class and are not modified before use. The second most diffused category is CUSTOMISE which indicates that often several methods are called to modify the method's argument. Extension patterns belonging to the EXTEND and IMPLEMENT categories are a minority. This result is similar to the one obtained in the reference paper [9]. It is remarkable that we do not identify any instances of the IMPLEMENT category. We believe this is because in Scala only traits are barely used as the first parent to inherit from. We also observe that the distributions are similar across all the frameworks.

D. *RQ_4 — How are extension points distributed across framework classes?*

Design. The recommendations provided by SCALA-XP-MINER are useful only if developers are not overloaded by suggestions. For this reason, we evaluate the distribution of extension points for each class. We group the extension graphs by the class of the receiver and then compute the number of distinct extension points for each class. As for $\mathbf{RQ_2}$, we report the results only for SPARK. The remaining results are contained in our online appendix [11].

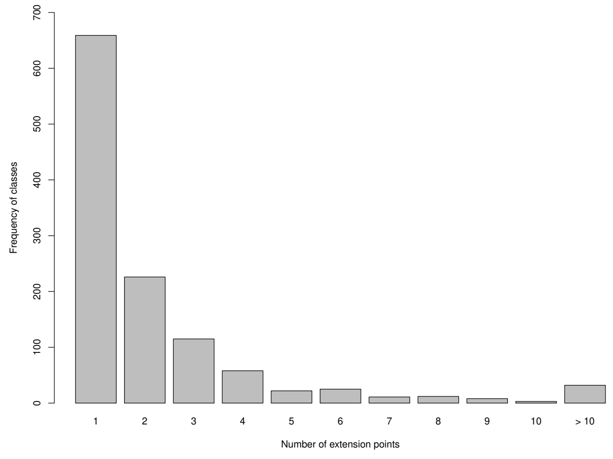

Fig. 5. Distribution of extension points by classes on SPARK

FEMIR Results. The results show that only 4% of the classes have 10 or more extension points. Thus, the suggestions can be read easily by developers.

SCALA-XP-MINER Results. Figure 5 depicts the distribution of extension points by classes (receiver type) on the SPARK framework. Similar to the reference paper [9], we observe that the majority of the classes have less than four

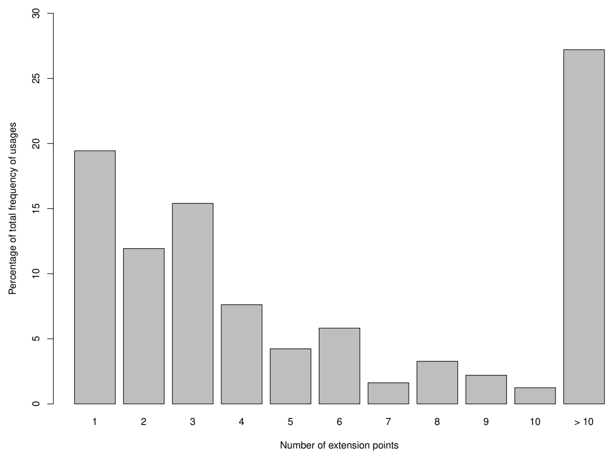

Fig. 6. Usage frequencies of SPARK framework classes with different numbers of extension points

extension points, while only a small number of the classes 3% have ten or more extension points. This distribution, similar for the remaining frameworks, means that it will rarely happen for a developer to be overloaded by a high number of suggestions. From fig. 6, however, we note that a small number of classes is highly used by developers. Indeed, 28% of the usages relies on these classes. However, this behaviour is not observable for all frameworks (*e.g.,* AKKA and PLAY).

E. *RQ_5 — To what extent do developers use multiple extension points to extend a component?*

Design. In this research question, our aim is to understand how frequently extension patterns are used together. To answer this question, we group the extension point usages, represented by extension graphs, based on the methods that are called on the same receiver object. For each of these methods we check if it corresponds to an extension point. Each group of extension graphs provides the set of extension points that are used together.

TABLE IV
NUMBER OF CASES IN WHICH MULTIPLE EXTENSION POINTS ARE USED TOGETHER

Framework	**1 (%)**	**2 (%)**	**3 (%)**	**4 (%)**	**>4 (%)**
Spark	16,812 (99.96)	4 (0.02)	1 (0.01)	-	1 (0.01)
Akka	6,821 (99.98)	2 (0.02)	-	-	-
Hadoop	1,676 (100.00)	-	-	-	-
Mockito	3,077 (100.00)	-	-	-	-
Play	7,283 (99.90)	3 (0.04)	4 (0.05)	-	1 (0.01)

FEMIR Results. In 83% of all extension points, developers use only one extension point to extend a component of a framework. Cases in which developers use five or more extension points together are rare.

SCALA-XP-MINER Results. Table IV reports the number of cases in which multiple extension points were used together, along with the percentages. As in the reference work [9], we find that rarely extension patterns are used together. Indeed,

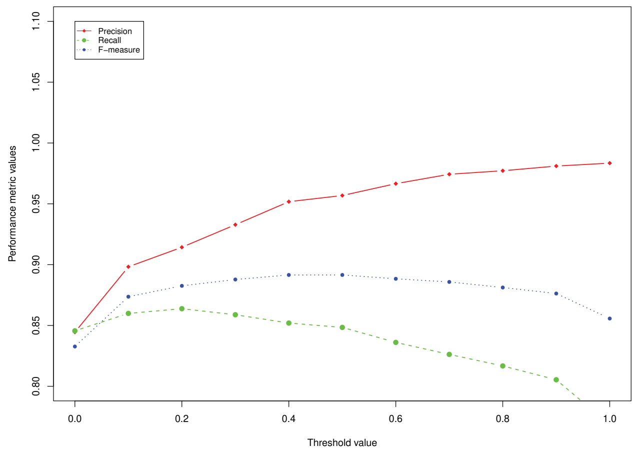

Fig. 7. Influence of the threshold on precision, recall and F-Measure for the SPARK framework.

the percentage of cases in which extension patterns are used together is less than 1%. We observed this behaviour also for other frameworks. In particular, in two cases (*i.e.,* HADOOP and MOCKITO) extension points are even never used together.

F. RQ$_6$ — What is the impact of algorithm tuning on the performance?

Design. We evaluate the influence of the threshold value δ on the results. As for **RQ**$_2$ and **RQ**$_4$, we show only the results for SPARK, because it is the largest framework considered in our replication study. Similar to **RQ**$_1$ and **RQ**$_2$, we use the 10-fold cross validation to determine *precision*, *recall*, and *F-Measure*, while varying the threshold value δ between 0 and 1 at intervals of 0.10. Considering that in the reference paper [9] it was not clear how many recommendations where selected, we replicated this research question taking the top-1 recommendation.

FEMIR Results. FEMIR-GLOBAL and FEMIR-LOCAL use a threshold value δ to improve the quality of the extension pattens. The authors evaluated the effect of tuning this value. They showed that the model built with $\delta = 0.3$ is able to achieve the best trade-off in terms of precision and recall.

SCALA-XP-MINER Results. Figure 7 depicts the variation of *precision*, *recall*, and *F-measure* as the value of the δ parameter increases in the case of SPARK. Our goal is to find a threshold that maximises the *F-measure*, as this measure represents a trade-off between *precision* and *recall*. From the plot it is clear that the *F-measure* reaches its highest value when the δ ranges between 0.3 and 0.5. After this point, further increasing δ gradually decreases the *F-measure*. For this reason, in contrast to the reference paper, we recommend a slightly higher threshold value of 0.4. We argue that this is due to the distribution of extension points by categories. Indeed, as seen in **RQ**$_3$, the majority of extension patterns belong to the SIMPLE and CUSTOMISE categories. Thus, considering the small average size of the patterns recommended by SCALA-XP-MINER, a slightly higher δ is suggested.

G. RQ$_7$ — How does the approach perform in terms of execution time?

Design. We measure the runtime performance of SCALA-XP-MINER by measuring the average time required to make recommendations for a given extension point. In particular, we calculate the execution time needed by SCALA-XP-MINER for mining the extension patterns and for providing a recommendation. As previously done for other RQs, we show the results obtained on SPARK, our largest framework. In total, we computed the execution time needed to recommend each one of the 1.093 recommendations and we aggregate the values using the *mean* operator. The mining process was performed on a machine with 2 Intel Xeon 2637 (4 cores at 3.50 GHz) and 256 GB of RAM, while the recommendations were analysed on a machine with an Intel CORE i7 (2.93 GHz) and 16 GB of RAM (similar to the one used as in the reference work [9]).

FEMIR Results. On average, recommending an extension graph requires $0.92s$. However, it is worth noting that most of the computational effort is needed for analysing the source code and building the extension graphs (*e.g.,* $78h$ for the JFACE project).

SCALA-XP-MINER Results. To compute a recommendation for SPARK, 0.24 seconds are needed on average. However, this is only a fraction of the total time because the time required to analyse the source code and to generate framework extension graphs is significantly higher (*i.e.,* about 17 hours for SPARK). It is worth mentioning that this is a one-time operation. Our results are consistent with those achieved in the reference work [9].

V. THREATS TO VALIDITY

In this section, we discuss the threats that might affect the validity of our replication study.

Threats to construct validity. We collected extension points in a large corpus of Scala projects using SCALA-XP-MINER. We are aware that the precision of SCALA-XP-MINER and the automated tool used for the evaluation have a crucial influence on the results. To mitigate this threat, we manually analysed a sample of the extension patterns to verify whether they were correctly detected. We identify SCALA-META and SEMANTICDB as threats to construct validity, given that SCALA-XP-MINER extensively uses these libraries which are relatively new. We were not able to resolve the type of every argument (*e.g.,* the result type of a polymorphic method call) in 6.47% of the cases. Nevertheless, these libraries have already been adopted by many industrial developers as indicated by the many use cases[3].

Threats to external validity. We selected a subset of the most used projects from the largest open-source hosting service GITHUB. We concluded that SPARK, HADOOP, AKKA, PLAY and MOCKITO are among the most used frameworks, while having different domains and characteristics. Considering that we rely on SCALA-META and SEMANTICDB, we

[3]https://scalameta.org/

selected only those projects built using SBT 0.13+ and Scala 2.11+. The selection of the subset could be a threat to external validity because these frameworks may not be directly applicable to industrial environments. To select the projects, we used the number of stars as a proxy for importance; we are aware that other proxies [13], [14] could be used, but our goal was to analyse those ones that were more popular by developers.

Threats to conclusion validity. We adopted 10-fold cross validation. We are aware of the existence of other validation methodologies that might provide a better interpretation of the results. However, we chose to apply this methodology to compare our results with those achieved by Asaduzzaman *et al.* [9]. The metrics (*i.e.,* precision, recall, and F-measure) employed to evaluate the accuracy of the extension patterns could form another threat. We relied on the same metrics as used in the reference work in order to compare results.

VI. Related Work

In addition to the work of Asaduzzaman*et al.* [9], the subject of our replication study, this section discusses related work in the areas of mining for patterns in framework extension and API usages.

Mining Framework Extensions. Michail [15], [16] developed CodeWeb, a technique that extracts reuse relationships to discover library usage examples. Bruch *et al.* [17] proposed FrUiT, an Eclipse plug-in relying on data mining techniques that extracts reuse patterns from existing framework instantiations to create framework usage scenarios based on five class properties (extends, implements, overrides, calls, and instantiations). Dagenais and Ossher proposed XFinder [18] to semi-automatically locate framework extension examples through concern-oriented documentation, so-called guides. This tool is close to Femir, but it requires developers to provide framework-specific steps (*e.g.,* extend class `A`, override method `m`, *etc.*) for each pattern in order to automatically locate implementation examples. In other words, the framework documentation is used as a template and XFinder finds instances of the template within the codebase. In contrast, Femir and scala-xp-miner are able to mine these patterns automatically based on implementation examples. Thummalapenta and Xie [19], [20] developed SpotWeb, a code search engine for frameworks and libraries written in Java. This tool determines the frequency of framework classes and methods by mining code examples. SpotWeb assists software developers in reusing APIs of an existing framework by detecting hotspots (*i.e.,* frequently used APIs) and coldspots (*i.e.,* barely used APIs). Bruch *et al.* [21] proposed an approach to document object-oriented white-box frameworks by mining four kinds of documentation items. Moritz *et al.* developed ExPort [22] to automatically mine and visualise API usage examples. This technique is able find API usage examples that occur across several functions of a program.

Mining API Usage Patterns. Archarya *et al.* [23] analyzed static traces of source code to mine API usage patterns as partial orders. Zhong *et al.* proposed MAPO [24]; this tool first clusters API calls, after which it generates sequences of method calls that are mined to discover frequent patterns. Considering that the sequences could be redundant, MAPO could produce redundant patterns. Nguyen *et al.* [8] introduced the Groum representation and proposed GrouMiner, a tool to mine frequent patterns and anomalous API usages from a dataset of Groums. Based on this work, Mover *et al.* [25] proposed an optimization that scales to large corpora of software systems. The optimization uses a combination of frequent itemset mining and SAT solving and was implemented in a tool called BigGroum.

Wang *et al.* [26] proposed two quality metrics (*i.e.,* succinctness and coverage). They employed these metrics in an approach that combines frequent pattern mining and clustering. The technique resulted in Up-Miner (Usage Pattern Miner), a tool capable of outperforming MAPO [24].

Afterwards, Fowkes *et al.* [27] proposed a Probabilistic API Miner (PAM) to mine API usage patterns that both outperforms MAPO and Up-Miner. Finally, Nguyen *et al.* presented FOCUS, a tool that mines open-source project repositories to recommend API usage patterns. FOCUS outperforms PAM in terms of success rate, accuracy, and execution time.

VII. Conclusion

In this paper, we presented a differentiated replication study [10] of the work by Asaduzzaman *et al.* [9] that proposes an approach to recommend extension patterns by mining Java source code examples. We replicated the study for frameworks and systems developed in Scala to verify whether the approach is also suitable in this context. To this end, we have proposed scala-xp-miner, a tool that analyses Scala projects, mines extension patterns, and recommends usable patterns for a given framework.

The results show that scala-xp-miner and thus the reference approach are able to propose accurate recommendations, even if in most cases these suggestions are rather simple. In particular, we find that the extension patterns recommended by scala-xp-miner on Scala are on average composed of three or four nodes which indicates that most framework usages are rather simple. This implies that the results obtained for Scala frameworks through scala-xp-miner are more accurate with respect to those obtained for Java ones through Femir. Extension points belonging to the Extend categories are rare, while the most common ones belong to the Simple and Customise categories. It is remarkable that we were not able to find any extension points belonging to the Implement category. This observation may be due to how Scala developers usually extend frameworks. Furthermore, some of the Scala features such as implicit parameters, companion objects, and by-name parameters enable Scala-specific means of using framework extension points. Supporting Scala-specific usages is part of our future agenda. We find that the distribution of the extension points by classes is similar to the reference work: most classes have less than four extension points. Despite this, 28% of the usages belong to the few classes that have more than ten extension points. We find that developers are unlikely to use multiple extension points together (less than 1% of the cases). Considering that in the reference work between 3% and 16% of extension points are used together, we state that this could be due to the selection of the frameworks or to the Scala language. Further analysis is part of our future agenda. Regarding the $delta$-parameter tuning, we find that a value of 0.4 achieves a better *precision*, *recall*, and *F-measure*. This value is slightly higher than the one for the reference work (*i.e.,* 0.3). The execution time needed to compute a recommendation is slightly lower ($0.24s$) with respect to the reference work. However, as concluded by Asaduzzaman *et al.* [9], most of the computational effort is due to the preceding source code analysis and pattern mining.

As future work, we first plan to implement a recommendation component, in the form of a plugin for ECLIPSE or INTELLIJ, to recommend extension points to developers. This plugin could assist not only researchers, but also developers in extending frameworks. Another direction will be to identify and support Scala-specific extension point usage means. This is likely to result in new categories of extension patterns that can be added to the taxonomy of the reference paper. Their incorporation in SCALA-XP-MINER might result in less straightforward pattern recommendations. With respect to this aspect, we will involve Scala developers in empirical studies where applying this tool on real systems could assess the patterns' usefulness and the real impact of the recommendations. Finally, we are also interested in extending SCALA-XP-MINER to assist developers when designing new frameworks. For instance, the tool may indicate what kinds of extension points are used infrequently, which extension points are more error-prone, or which ones more complex to use [28].

REFERENCES

[1] M. P. Robillard, "What makes apis hard to learn? answers from developers," *IEEE software*, vol. 26, no. 6, pp. 27–34, 2009.

[2] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated api property inference techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, 2013.

[3] C. De Roover, R. Lämmel, and E. Pek, "Multi-dimensional exploration of api usage," in *Proceedings of the 21st IEEE International Conference on Program Comprehension*, 2013.

[4] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: Helping to navigate the api jungle," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, 2005.

[5] S. Thummalapenta and T. Xie, "Parseweb: a programmer assistant for reusing open source code on the web," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 204–213.

[6] R. Alur, P. Černỳ, P. Madhusudan, and W. Nam, "Synthesis of interface specifications for java classes," *ACM SIGPLAN Notices*, vol. 40, no. 1, pp. 98–109, 2005.

[7] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *Proceedings of the 23rd European Conference on Object-Oriented Programming*, 2009, pp. 318–343.

[8] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 383–392.

[9] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "Recommending framework extension examples," in *2017 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2017, pp. 456–466.

[10] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[11] Y. Pacheco, J. De Bleser, T. Molderez, D. Di Nucci, W. De Meuter, and C. De Roover. (2019) Mining scala framework extensions for recommendation pattern. https://figshare.com/projects/saner2019-appendix/58529. Online appendix.

[12] C. C. Aggarwal, *Data mining: the textbook*. Springer, 2015.

[13] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of github repositories," in *Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2016, pp. 334–344.

[14] M. Nagappan, T. Zimmermann, and C. Bird, "Diversity in software engineering research," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 466–476.

[15] A. Michail, "Data mining library reuse patterns in user-selected applications," in *ase*. IEEE, 1999, p. 24.

[16] ——, "Data mining library reuse patterns using generalized association rules," in *Proceedings of the 22nd international conference on Software engineering*. ACM, 2000, pp. 167–176.

[17] M. Bruch, T. Schäfer, and M. Mezini, "Fruit: Ide support for framework understanding," in *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*. ACM, 2006, pp. 55–59.

[18] B. Dagenais and H. Ossher, "Automatically locating framework extension examples," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 203–213.

[19] 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*. IEEE Computer Society, 2008, pp. 327–336.

[20] ——, "Spotweb: detecting framework hotspots via mining open source repositories on the web," in *Proceedings of the 2008 international working conference on Mining software repositories*. ACM, 2008, pp. 109–112.

[21] M. Bruch, M. Mezini, and M. Monperrus, "Mining subclassing directives to improve framework reuse," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 141–150.

[22] E. Moritz, M. Linares-Vásquez, D. Poshyvanyk, M. Grechanik, C. McMillan, and M. Gethers, "Export: Detecting and visualizing api usages in large source code repositories," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2013, pp. 646–651.

[23] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining api patterns as partial orders from source code: from usage scenarios to specifications," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 25–34.

[24] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *European Conference on Object-Oriented Programming*. Springer, 2009, pp. 318–343.

[25] S. Mover, S. Sankaranarayanan, R. B. P. Olsen, and B.-Y. E. Chang, "Mining framework usage graphs from app corpora," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2018, pp. 277–289.

[26] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage api usage patterns from source code," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 319–328.

[27] J. Fowkes and C. Sutton, "Parameter-free probabilistic api mining across github," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 254–265.

[28] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, and M. Di Penta, "Focus: A recommender system for mining api function calls and usage patterns," in *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering*, 2019.