A Cooperative Approach for Combining Client-based and Library-based API Usage Pattern Mining

Mohamed Aymen Saied, Houari Sahraoui
DIRO, Université de Montréal, Montréal, Canada
{saiedmoh,sahraouh}@iro.umontreal.ca

***Abstract*—Software developers need to cope with the complexity of Application Programming Interfaces (APIs) of external libraries or frameworks. Typical APIs provide thousands of methods to their client programs, and these methods are not used independently of each other. Much existing work has provided different techniques to mine API usage patterns based on client programs in order to help developers understanding and using existing libraries. Other techniques propose to overcome the strong constraint of clients' dependency and infer API usage patterns only using the library source code. In this paper, we propose a cooperative usage pattern mining technique (COUPminer) that combines client-based and library-based usage pattern mining. We evaluated our technique through four APIs and the obtained results show that the cooperative approach allows taking advantage at the same time from the precision of client-based technique and from the generalizability of library-based techniques.**

***Keywords*-API Documentation; API Usage; Usage Pattern; Software Clustering;**

I. Introduction

Despite recent progress in API documentation and discovery [1]–[3], large APIs are still difficult to learn and use [2], [4]. A large API may consist of several thousands public methods defined in hundreds classes. The API methods are generally used within client programs jointly with other methods of the API, but it is not evident to deduce the co-usage relationships between API methods from their documentations. To address this issue, much research effort has been dedicated, recently, to the identification of API usage patterns [1], [2], [5], [6]. A large part of the existing work in API usage patterns mining relies on client programs' code to identify common ways to use the API. For example, in [6] we proposed a technique for mining Multi-Level API Usage Patterns (MLUP) to exhibit the co-usage relationships between methods of the API of interest across interfering usage scenarios. This technique was based on the analysis of the frequency and consistency of co-usage relations between the APIs methods within a variety of client programs of the considered API. The client-based techniques are known for their accuracy. However, client programs' code is unfortunately not available for both newly released libraries and APIs that are not widely used. And even if client programs are available, in an exhaustive scenario, those clients may not cover all the possible usage contexts of the API of interest.

To address these issues, other research contributions [7], [8] overcome the strong constraint of client dependency by only considering the library code. Indeed, in our recent work [8], we proposed a technique for inferring Non Client-based Usage Patterns (NCBUPminer). This technique is based on the idea that API methods can be grouped into usage patterns by considering their mutual relationships. Our intuition is that related methods of the API may contribute together to the implementation of a domain functionality in client programs and thus may form an API usage pattern. NCBUPminer is premised on the analysis of structural and semantic dependencies of API methods within the library code itself.

Deriving the usage patterns from library code is interesting as the inferred patterns could apply to any client program (generality property). However, this kind of derivation could be unsafe since it could lead to inferring usage patterns that do not reflect a real-world behavior. Conversely, client-based derivation infers precise usage patterns with actual instances in the used client programs (accuracy property). However, these patterns are specific to the considered clients. In this paper, our objective is to take advantage of the properties of generality and accuracy by combining the client-based and library-based usage pattern mining. Our idea is similar to that of hybrid static-dynamic analysis, which aims at finding a good compromise between the static-analysis soundness and the dynamic-analysis accuracy [9].

We specifically study which form of combination is better suited to achieve the best tradeoff between the generality and accuracy properties. As both techniques follow an iterative mining process to refine the patterns, the obvious combination is the sequential one, by applying a first technique (client or non client-based mining) to derive a set of patterns and then apply the second technique to refine these patterns. A more sophisticated combination is to interleave the different iterations of the two techniques (starting by one or the other technique) in a parallel and cooperative manner to solve a common goal. As both techniques use parameters, the values of the parameters can be varied to improve the accuracy and to explore the search space more efficiently.

We evaluated our hybrid technique using four different APIs: HttpClient [10], Java Security [11], Swing [12] and AWT [13]. In our evaluation, we used 12 client programs of the HttpClient and Java Security APIs, and 22 client programs of the Swing and AWT APIs. The results indicate that the cooperative approach is better than the sequential one, and that it performs significantly better than three other single technique alternatives [6], [8], [14].

The remainder of the paper is organized as follows. Section II discuss the motivation behind our work. We explain our approach in Section III and we detail the cooperative mining in Section IV. The approach evaluation setting and the used APIs are described in Section V. Section VI presents and discuss the results of our study. Finally, Section VII presents various contributions that are related to our work, before concluding in Section VIII.

II. Background And Motivation Examples

In this section, we first present some definitions. Then, we illustrate through a motivating example how the combinations of client-based and library-based technique can lead to a better approximation of real-world behavior.

A. Background

Existing API usability and property inference research targets two types of API usage patterns [15]: (1) unordered usage patterns, and (2) sequential usage patterns. A pattern is considered to be a typical or common way to use an API. Patterns generally emerge from the usage data in contrast regular documentation, which is formally specified by a developer [15]. Unordered usage patterns refer to a set of API elements, generally public methods, observed to be used together with a certain frequency within a set of usage contexts. Sequential usage patterns differ from the unordered ones in that they consider the order in which API elements are used [15]. For example, for an Input/Output API, a possible usage pattern that can be observed is open;close which indicates that whenever client code calls an API method open, it also calls close, and vice versa. A mined sequential pattern would add the property that open should precede close [15]. Clearly, mining sequential patterns is more challenging. However, for usage patterns composed of many API methods, unordered patterns are a good starting point for client program developers. In this paper, we are interested in unordered usage patterns, and we use the term ”usage patterns” to refer to them.

When inferring API usage patterns, the goal is to capture as much as possible the real-world behavior. We want to cover the majority of potential cases, and we need to be as accurate as possible. However, existing techniques have to choose between the generality, when usage patterns are derived from the library code, or the accuracy, when the patterns are mined by analyzing the interaction between client programs and the library. To explain this dilemma, let us consider two of our previous contributions. In [6], we propose a client-based technique for mining *multi-level API usage patterns*. Despite the accuracy of this technique, it was unable to cover some usage contexts documented in the API. In another work [8], we propose a technique for inferring Non Client-based Usage Patterns (NCBUPminer). This technique allowed us to approximate a large number of patterns from which many are not accurate.

B. Java Security Example

Digital signatures are used for authentication and integrity assurance of digital data. The `Signature` class of the Java Security API [11] is used to provide this functionality. A `Signature` object can be used either for signing data or for verifying digital signatures. For that, a client program usually starts by getting the `Signature` object that implements a specified standard signature algorithm through the method `getInstance(String algorithm)` in the `Signature` class. Then, one needs to initialize the `Signature` object either with a private key for signing (`initSign(PrivateKey)`), or with a public key for verification (`initVerify(PublicKey)`). In the next step, the `update(byte[] )` method is used to update the data to be signed or verified. Finally, depending on the initialization type, `sign()` method is used for signing the updated data or `verify(byte[] signature)` is used to verify the passed-in signature. Let us, now, show how such a pattern has been mined using the client-based and library-based techniques.

The client-based technique [6] detected a usage pattern reflecting the integrity assurance functionality through the following 4 API methods `getInstance(String algorithm)`, `initSign(PrivateKey)`, `sign()` and `update(byte[])`. The client-based technique missed the methods related to verifying digital signatures, since these methods were not sufficiently used by the considered client programs.

The library-based technique [8] grouped 12 API methods in a pattern reflecting the integrity assurance functionality. The pattern includes the following methods: `initVerify(PublicKey)`, `initSign(PrivateKey)`, `sign()`, and `verify(byte[] signature)` in `Signature` class; `generatePublic(..)`, `generatePrivate(..)` and `getInstance(..)` in `KeyFactory` class; `getInstance(..)`, `initialize(..)` and `generateKeyPair()` in `KeyPairGenerator` class; `getPrivate()` and `getPublic()` in `KeyPair` class. Although this pattern includes the digital signature verification functionality, some of the associated methods, e.g., generation of pairs of public and private keys, are not always relevant.

We will show later in this paper that combining both client and non-client based techniques would help us restricting the mined pattern to only the 8 necessary API methods: `getInstance(String algorithm)`, `initSign(PrivateKey)`, `sign()`, `initVerify(PublicKey)`, `verify(byte[] signature)`, `update(byte[] )` in `Signature` class and `generatePublic(..)`,`generatePrivate(..)` in `KeyFactory` class.

III. Usage Patterns Analysis

This section introduces our cooperative approach for detecting API usage patterns. Before presenting the used algorithm, we provide a brief overview of our approach and the representation of usage patterns in our approach.

A. Overview

We define a usage pattern for an API as a subset of the API's methods that are either co-used together by the API client programs, or structurally and semantically related and

thus may jointly contribute to the implementation of a domain functionality in client programs. A usage pattern includes only public API methods that can be accessed from client programs, and each pattern represents an exclusive subset of the APIs methods.

Ideally, the co-usage relations between the pattern's methods remain the same for all possible clients. However, APIs are open applications, and it is unfeasible to analyze all their possible usage scenarios (clients). For this reason, client-based mining emphasizes patterns' accuracy over their generality. Conversely, for the library-based mining, public methods that change the state of or manipulate the same set of objects while sharing the same semantic context, could cooperate to accomplish certain domain functionality. This allows to find general patterns independently from a given subset of clients. However, those methods, although similar, may not represent an actual usage pattern in practice. Hence, the library-based mining emphasizes patterns' generality over their accuracy. To benefit from both generality and accuracy properties, our approach combines both client-based and library-based techniques.

Our approach takes as input the source code of the API to study and multiple client programs making use of this API. The output is a set of usage patterns as described earlier. The detection approach follows three steps:

- *Extracting API methods, references, terms and usage.* First, the API source code is statically analyzed, and its public methods are extracted. We collect, for each public API method, all the fields that are referenced either directly inside it or through the methods that it uses. We also collect terms composing the public method name and its parameters as well as the local variable identifiers. In addition to fetching information inside the library, all the provided client programs are statically analyzed to extract the occurrences of API methods that are classed in the clients' methods. The static analysis is performed using the Eclipse Java Development Tools (JDT).
- *Encoding methods information.* After the information extraction, we derive states, terms and usage vectors for the API public methods. Each public method in the API is characterized by: (1) a state vector which encodes information about objects and states manipulated by the method, (2) a vector of the method's terms, that will be used by LSI technique [16], [17] to construct a semantic space representation for the API of interest, and (3) a vector of the method's usage which encodes information about its client methods.
- *Clustering.* Finally, we use cluster analysis to group the API methods that have the best trade-off between structural, semantics and co-usage relationships.

B. Information Encoding of API Methods

In our approach, an API public method represents a point in the search space. As mentioned above, each point is represented by three vectors. The states' vector has a constant length, which is the number of all the manipulated classes and fields through the API public methods. For a given API method, an entry of 1 (resp. 0) in the i^{th} position of this vector, denotes that the i^{th} field is referenced (resp. not referenced) in the API method. If at least one field of a class is referenced, then the vector element corresponding to the class is set to 1. The classes are also considered because, when computing the state similarity between two methods, more importance is given to situations where both methods access fields from the same class than from different classes.

The terms' vector is computed from the lemmatized collected terms (see Section III-A). Similarly to states' vectors, the terms' vectors have a constant length that is the number of all lemmatized terms collected from the public methods in the API of interest. For a given API method, an entry of 1 (resp. 0) in the i^{th} position of the terms' vector, denotes that the i^{th} term appears (resp. does not appear) in the method vocabulary. The terms' vectors of the API are used in a Latent Semantic Indexing process [16] to create a term-document matrix C where each row represents a term, and each column a document,i.e.,, an API public method in our case. This matrix is latter converted into a document matrix to capture the semantic similarity relations based on terms co-occurrence, following the Singular Value Decomposition (SVD) [16] technique. For more details about this derivation, we refer the reader to [8].

The usage vector, also has a constant length, which is the number of all clients' methods that use the API methods. For an API method, m, an entry of 1 (or 0) in the i^{th} position of its usage vector, denotes that m is used (or not used) by the i^{th} client method.

C. Similarity and Distance Metrics

As mentioned earlier, our approach constructs clusters of API methods by grouping those that are close to each other (i.e., similar methods) either from the client perspective (i.e., co-usage relationships) or from the library perspective (i.e., structural and semantic relationships). For this purpose, we define three similarity metrics, State Manipulation Similarity $StateSim$, Semantic Similarity $SemanticSim$ and Usage Similarity $UsagSim$.

The rationale behind $StateSim$ metric, as defined in Equation (1), is that two API methods m_i and m_j are close to each other if they share a large subset of the classes and fields they are manipulating.

$$StateSim(m_i, m_j) = \frac{|accessed(m_i) \cap accessed(m_j)|}{|accessed(m_i) \cup accessed(m_j)|} \quad (1)$$

For the semantic similarity metric, $SemanticSim$, we use the SVD document matrix, V_k^T, as mentioned above, to compute the cosine similarity between the API methods, as defined in Equation (2). This measure is used to determine how much relevant is the semantic information shared among two API methods.

$$SemanticSim(m_i, m_j) = \frac{\overrightarrow{V_i} \times \overrightarrow{V_j}}{||\overrightarrow{V_i}|| \times ||\overrightarrow{V_j}||} \quad (2)$$

Where $\overrightarrow{V_i}$ and $\overrightarrow{V_j}$ are the i^{th} and j^{th} column corresponding to m_i and m_j in the document matrix V_k^T. The semantic similarity is then normalized between 0 and 1.

Regarding the usage similarity, $UsageSim$ as defined in Equation (3), the rationale is that two API methods are close to each other if the corresponding methods share a large subset of common client methods.

$$UsageSim(m_i, m_j) = \frac{|Cl\_mtd(m_i) \cap Cl\_mtd(m_j)|}{|Cl\_mtd(m_i) \cup Cl\_mtd(m_j)|} \quad (3)$$

Where $Cl\_mtd(m)$ is the set of client methods of the API method m.

Using the similarity metrics, we compute the client based distance C_Dist and the library based L_Dist distance between two points p_i and p_j representing respectively two API methods m_i and m_j as follows:

$$C\_Dist_{ij} = 1 - UsageSim(m_i, m_j) \quad (4)$$

$$L\_Dist_{ij} = 1 - \frac{StateSim(m_i, m_j) + SemanticSim(m_i, m_j)}{2} \quad (5)$$

D. Clustering algorithm

Our clustering is based on the algorithm DBSCAN [18]. DBSCAN is a density based algorithm, i.e., the clusters are formed by recognizing dense regions of points in the search space. The main idea behind DBSCAN is that each point to be clustered must have at least a minimum number of points in its neighborhood. This property of DBSCAN permits to leave out (not cluster) any noisy point that is not located in a dense region of points in the search space.

DBSCAN depends upon two parameters to perform the clustering. The first parameter is the minimum number of methods in a cluster. In our context, we set it at two, so that a usage pattern must include at least two methods of the studied API. The second parameter, epsilon, is the maximum distance that within which two methods can be considered as neighbors. In our case, we use the client-based distance Equation (4) when the clustering is performed from the client perspective, and the library-based distance Equation (5) when the clustering is performed from the library perspective. In practice, the choice of the epsilon value is not straightforward. In our client-based, non-client-based and cooperative pattern mining techniques, we start with an initial value of epsilon to identify groups of very-close methods. For example, a value of 0 means that each cluster must contain only API methods that are completely similar. This results in a first level of patterns. Then, epsilon is increased to include new methods in the existing patterns. This process is repeated until all the methods are clustered or a fixed threshold for epsilon is reached. The final result is a set of multi-level usage patterns.

IV. Cooperative Patterns Mining

This section details the combination process of the cooperative API usage-pattern mining.

As mentioned before, the API public methods represent points in the search space. These points will be cooperatively clustered from two different perspectives. (i.e., client-based and library-based), which allows examining the search space more efficiently. The challenge is then, how to combine these two perspectives to achieve their respective benefits. For this question, we have two possibilities: (1) the two techniques are ran one after the other (sequential combination), or (2) they are intertwined (parallel combination).

A. Cooperative sequential combination

As mentioned earlier, the client-based and the library-based mining could be performed one after the other. The rationale behind the sequential combination is that the obtained usage patterns with one mining technique can be completed/enriched by the other technique. For the sequential combination, we will test the possibilities of starting with one or the other technique.

1) Start with the client-based mining: In this case, as shown in Figure 1, we start the combined process with the client-based mining, and the clustering is performed using the client based distance Equation (4). At the beginning, we construct a dataset containing all the API methods and cluster them using DBSCAN algorithm with an epsilon value close to 0. This results in clusters of API methods that are always used together. The other methods are considered as noisy points. After this run, for each produced cluster, we aggregate the usage vectors of its methods using the logical disjunction into one vector to form a new dataset. The new dataset includes the aggregated usage vectors and the usage vectors of noisy methods from the first run. This dataset is fed back to the clustering algorithm, but with a slightly higher value of epsilon. This procedure is repeated until epsilon reaches a maximum threshold value.

At this point, we switch to the library-based mining. For that, all the resulting clusters and noisy points are used to construct the input dataset. The clustering is then performed using the library-based distance Equation (5). Here again, we start by an epsilon value of 0. Then, we repeat the process with the new technique: (1) clustering, (2) dataset updating, and (3) epsilon slight increase, until we reach a second epsilon threshold. To update the dataset, the state and term vectors of the already-clustered methods are respectively aggregated using the logical disjunction. The new term vector $\overrightarrow{T}$ is then mapped into its representation in the LSI semantic space by the following transformation:

$$\overrightarrow{T_k} = \Sigma_k^{-1} \times U_k^T \times \overrightarrow{T} \quad (6)$$

2) Start with the library-based mining: For this second sequential combination possibility, we start with the library-based mining, and then the resulting clusters and noisy points are used to construct the input dataset for the client-based mining, which is repeated until epsilon reaches a maximum value given as a parameter.

B. Cooperative parallel combination

For the parallel combination, the client-based and the library-based mining are intertwined. The two mining process will, step by step, incrementally evolve in parallel exchanging the clustered data. In each step, either an iteration of the client-based mining or an iteration of the library-based mining is performed. Two epsilon values are maintained in parallel one for each technique: epsilonC and epsilonL. In the parallel combination, we can choose to start with the client-based mining

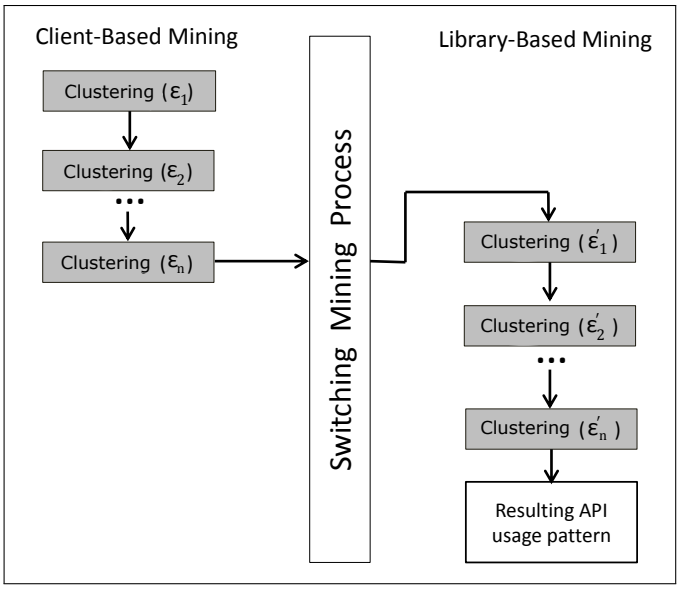

Figure 1. Sequential combination start with the client-based mining

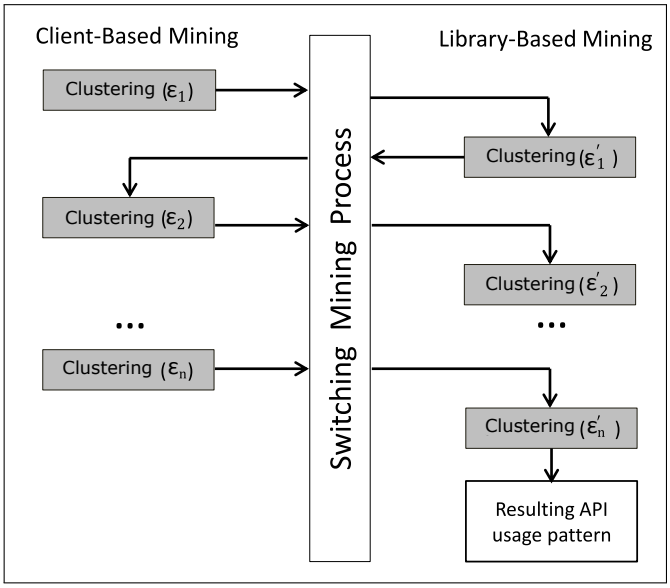

Figure 2. Parallel combination start with the client-based mining

or the library-based mining. The rationale behind the parallel combination is to go along the same lines of incremental clustering allowing the data to be mutually influenced from the beginning of the pattern construction.

1) Start with the client-based mining: In this case, as shown in Figure 2, we start the parallel combination with the client-based mining, and the clustering is performed using the client based distance Equation (4). At the beginning, all the API methods are clustered using DBSCAN algorithm with epsilonC value close to 0. At the end of this run, the mining process is switched after updating the dataset according to the resulting clusters and the noisy methods. As for the next iteration we use the library-based clustering, the dataset is updated by considering the state and term vectors, and the clustering is performed using the library-based distance Equation (5) with an epsilonL value of 0. At the end of this iteration, we switch back to the client-based clustering. The dataset is updated by the aggregation of the usage vectors, and epsilonC value is slightly increased. This cooperative process is repeated until epsilonC or epsilonL reach a value given as a parameter.

2) Start with the library-based mining: In this case, we reproduce the previous mining processus with the difference that we start the parallel combination with the library-based mining.

V. Evaluation

The objective of the evaluation study is to assess whether a cooperative approach can lead to a better inference of API usage patterns compared to individual client-based and library-based techniques, in terms of the tradeoff between the generality and accuracy properties of the mined patterns. The evaluation is performed in two steps. We first determine which of the four combination options leads to the best tradeoff. Then, we compare the cooperative approach, i.e., the best combination option, with individual techniques. We formulate the research questions of our study as follows:

- **RQ1:** what is the best strategy to combine the client-based and the library-based mining, and what is the impact of the different strategies on the quality of the inferred API usage patterns from the perspective of client programs?
- **RQ2:** to what extent does the cooperative parallel approach performs better than the individual techniques?

A. Comparative Evaluation

In this paper, we study the benefits of the cooperative approach (COUPminer) with respect to individual techniques, including the comparison with techniques proposed by other researchers. In our previous contributions, we have shown that the client-based technique (MLUPminer) [6] and the library-based technique (NCBUPminer) [8] perform better than the state-of-the-art technique MAPO [2]. Thus, we limited our evaluation to the comparison with MLUPminer and NCBUPminer.

To be consistent with the previous comparison studies, the evaluation in this paper is performed on the same four well-known APIs[1]: HttpClient, Java Security, Swing and AWT (Table I). The APIs are considered together with 22 client programs for the Swing and AWT APIs, and 12 client programs for the HttpClient and Java Security APIs .

To address the first research question, **RQ1**, we apply each combination strategy for detecting patterns in the considered APIs. Then, we compare the quality of detected usage patterns in the contexts of the selected client programs, using the parameters and metrics that we detail in Section V-B. We use the Wilcoxon rank sum test with a 95% confidence level to assess whether a significant difference exists between the measurements of the four combination strategies.

[1]We used the same client programs selected for the study of (MLUPminer and NCBUPminer). The list of programs and their descriptions is available in [8].

Table I
SELECTED APIS FOR THE CASE STUDY

API	Description
Java Security	Provides features to improve security of Java applications
HttpClient	Implements standards for accessing resources via HTTP
Swing	An API providing a graphical user interface (GUI) for Java programs
AWT	An API for providing a platform dependent graphical user interface (GUI) for a Java program

To address our second research question, **RQ2**, we compare the best cooperative strategy (COUPminer) with the client-based (MLUPminer) [6] and library-based (NCBUPminer) [8] techniques.

B. Experimental Setup

This section describes the used metrics in our study, as well as the setting and process of the performed experiments.

To assess the quality of the detected API usage patterns from the perspective of the API client programs, we need to evaluate whether these patterns are cohesive enough to exhibit informative co-usage relationships between the API methods. To measure the usage cohesion of the detected patterns, we use the Pattern Usage Cohesion Metric (PUC). PUC was originally proposed and used in [19] for assessing the usage cohesion of a service interface. It evaluates the co-usage uniformity of an ensemble of entities, in our context, a group of API methods, which forms a pattern. PUC states that a usage pattern has an ideal usage cohesion (PUC = 1) if all the pattern's methods are actually always used together. PUC takes its value in the range [0..1]. The larger the value of PUC is, the better the usage cohesion is. The PUC for a given usage pattern *p* is defined as follows:

$$PUC(p) = \frac{\sum_{cm} ratio\_used\_mtds(p, cm)}{|CM(p)|} \quad (7)$$

where *cm* denotes a client method of the pattern *p*, that is defined in a client program of the API of interest; *ratio_used_mtds(p, cm)* is the ratio of methods that belong to the usage pattern *p* and are used by the client method *cm*; and $CM(p)$ is the set of all client methods of the methods in *p*.

1) RQ1: To assess the quality of inferred patterns using COUPminer, and analyze the impact of different combination heuristics, we run the cooperative technique four times on each studied API. Each run uses all the API's public methods as the dataset to be clustered. In the first and second run, we consider the sequential combination heuristic, whilst in the third and fourth runs, we consider the intertwined parallel combination heuristic. Each heuristic was run twice, once starting with the client-based mining, and once starting with the library-based mining. For each studied API, we collect the inferred API usage patterns for the four runs and analyze their quality w.r.t. their usage cohesion (i.e., PUC values) in the context of the API client programs selected for the study. Note that some of the patterns inferred by our technique, COUPminer, may not be covered by the selected client programs. Therefore, we collect and consider the usage cohesion only for eligible patterns. In our study, an API usage pattern is said eligible if at least one of its methods is used/covered by one of the analyzed API client programs. Based on the comparison results, we decide which strategy is the best for inferring API usage patterns.

2) RQ2: We address our second research question in two steps, as follows. In the first step, for all the selected APIs of the case study, we apply NCBUPminer, MLUPminer and the best cooperative strategy COUPminer. Then, we compare the number and average size of inferred patterns for each technique. In the second step, we perform leave-one-out cross-validations to assess whether COUPminer achieves a good tradeoff between library-based and client-based techniques.

The cross-validations was performed using the API's client programs selected for the study. Let N represents the number of used client programs for the considered API (e.g., N = 22 for Swing), we perform N runs of the three compared techniques (NCBUPminer, MLUPminer and COUPminer) on the API. Each run uses N-1 client programs as training client programs for detecting usage patterns, and leaves away one different API's client program for validation. Obviously for NCBUPminer both the training clients and the validation client only served to assess the patterns' quality, since NCBUPminer is a non client-based technique.

To ensure that experimental observations are due to the proposed heuristic and not to the parameter choice, we reused the same configuration for the three techniques, e.g., the maximum value of the epsilon parameter is set to 0.35.

Patterns Validation Cohesion: We used the results of the cross-validations to evaluate the cohesion of the detected usage patterns (as measured by PUC) in the contexts of validation sets. However, in a given run, it is possible that some detected usage patterns involve only methods that are not used at all in the validation client programs. Therefore, we consider only patterns that contain at least one method that is actually used by the run's validation client programs. For such a pattern, if only few of its methods are used by the validation client programs, whereas the other methods are not, the pattern will have a low usage cohesion. At the end of this step, we compare the cohesion results obtained with NCBUPminer, MLUPminer and COUPminer.

Patterns Consistency: For each run of the cross-validations, we evaluate the consistency of the detected usage patterns between the training client programs and the validation client program. A pattern is said consistent if the co-usage relationships between the pattern's methods in the context of the training client programs remain the same (or very similar) in the context of the validation client programs. We define the consistency of a usage pattern as:

$$Consistency(p) = 1 - |PUC_T(p) - PUC_V(p)| \quad (8)$$

Where PUC_T and PUC_V are the usage cohesion values of the pattern *p* in the training client programs context and validation client programs context, respectively. This metric takes its value in the range [0..1]. A value close to 1 indicates that the co-usage relationships between the pattern's methods remain the same between training client programs and the *new* validation client program, while a value close to 0 indicates

a dissimilar behavior of the pattern between the two sets of clients. This metric allows us to see whether, between changing contexts, good patterns remain good ones and bad patterns remain bad ones.

VI. Results and Discussion

In the following sections, we discuss the results of our experiments.

A. Impact of Used Combination Strategies (RQ1)

To answer RQ1, we inspect the impact/contribution of both the sequential combination heuristic and the intertwined parallel combination heuristic. For each of these two heuristics, we first need to compare the possible strategies, i.e., starting with the client-based mining and starting with the library-based mining.

As illustrated in Table II, for the sequential combination strategy, when the client based mining is considered first, the inferred usage patterns exhibit less-good co-usage relationships between the pattern's methods. The average usage cohesion values for this case are between 73% for Swing and 81% for Java Security. Nevertheless, when the library-based mining is considered first, the usage cohesion values of inferred patterns are improved for three out of four libraries (between 75% for Swing and 85% for Java Security). The only slight degradation was observed for HttpClient from 78% to 77%.

For the intertwined parallel combinations, we observed the same trend. When the process starts with the client-based mining, all the average usage cohesion values were higher ta 80%, except for Swing with 77%. This low value can be attributable to the large size of Swing. Indeed, Swing declares 7226 public methods, as compared to Java Security API, which declares only 901 public methods. Recall that the values for Swing also were the lowest for the sequential combination strategies. The best cohesion values are obtained for the cooperative strategy with library-based mining is used as the initial technique. In this case, the usage cohesion values improve to reach, in the worst case, 82% for HttpClient and in the best case 88% for Java Security.

For both the sequential and the intertwined parallel combinations, starting library-based mining was the best option. The mined patterns reveal that with this option, it is possible to retrieve the majority of potential patterns, then, the client-based mining allows refining these patterns. However, the refinement is not performed at the same scale for the two combination heuristics. It is more efficient at a low scale, when the refinement is performed at the early stage of the pattern inference, which is the case for the intertwined mining.

Indeed, Table II clearly show that, in average, the intertwined parallel combination outperforms the sequential combination for detecting cohesive usage patterns. These results are statistically significant according to the Wilcoxon rank sum test. The usage cohesion values obtained for intertwined parallel combination reflect strong co-usage relationships between the pattern's methods for all the studied APIs.

Table II
Average Cohesion of identified API usage patterns, for NCBUPminer, MLUPminer and COUPminer.

	SequentialCombination		ParallelCombination	
API	**Start with clientBased**	**Start with libraryBased**	**Start with clientBased**	**Start with libraryBased**
Security	0.81	0.85	0.85	0.88
HttpClient	0.78	0.77	0.80	0.82
Swing	0.73	0.75	0.77	0.83
AWT	0.77	0.79	0.82	0.85

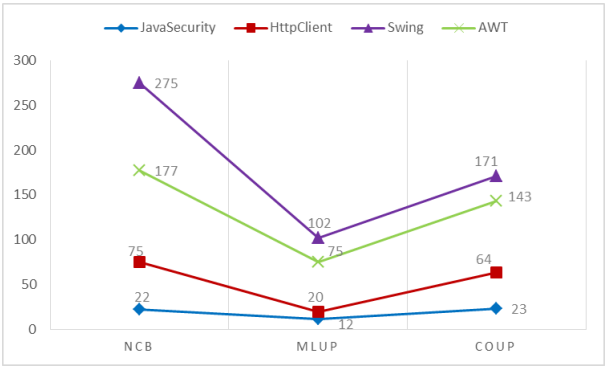

Figure 3. *Number* of inferred patterns using different techniques

B. Comparative Evaluation (RQ2)

To answer our research question RQ2, we applied NCBUPminer, MLUPminer and COUPminer, then we analyze the number and size of inferred API usage patterns, as well as their consistency and validation cohesion.

1) Number and Size of Inferred Patterns: Figure 3 shows that a trend can be observed between the numbers of patterns inferred through the three mining techniques. For all studied APIs, the lowest number of inferred patterns was obtained while only the client-based heuristic (MLUPminer) is used. Whereas the highest number of patterns was inferred when only the library-based heuristic (NCBUPminer) is used. We notice that a compromise is achieved while combining the library-based and client-based heuristics (COUPminer). For instance, in the case of AWT the number of pattern was adjusted to 143 while it was 75 with MLUPminer and 177 with NCBUPminer. The results in Figure 4 show that a similar trend can be observed for the patterns' size. The largest usage patterns were inferred when NCBUPminer is used, and the smallest patterns were inferred when MLUPminer is used. Once again, we notice that COUPminer finds a tradeoff in terms of patterns' size.

2) Cross-Validation : The cross-validation allows us to observe on two levels the effect of COUPminer on inferred patterns. First, we inspect the co-usage relationships of the detected patterns in the context of potential new client programs. Then, we characterize the usage cohesion deterioration between the training and validation context.

Patterns Validation Cohesion: As it can be seen in Table III, with MLUPminer, we obtained the highest average validation cohesion with around 83%. This indicates very strong co-usage relationships within the inferred patterns. In the case of NCBUPminer, we notice a visible decline in the

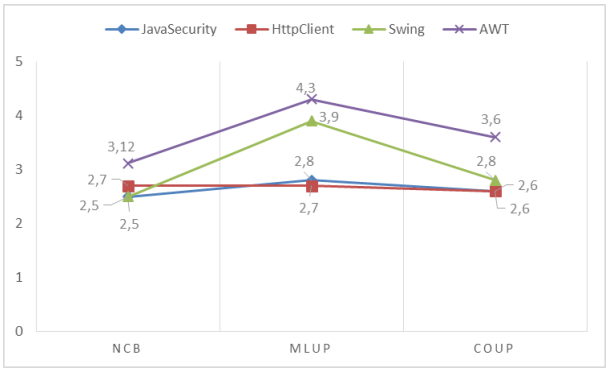

Figure 4. Average *size* of inferred patterns using different techniques

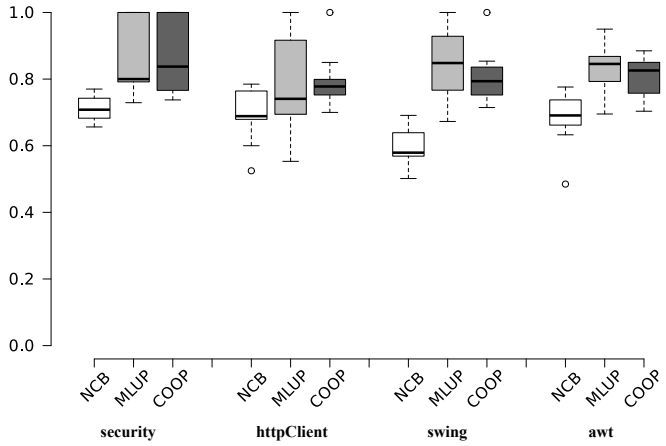

Figure 5. Cohesion values of identified API usage patterns, for NCBUPminer, MLUPminer and COUPminer in the contexts of validation clients

validation cohesion value with an average validation cohesion around 67%. In the case of COUPminer the decline was very limited with average values around 81%. We also notice that the standard deviation values are very low. This shows that, overall, the detected patterns using COUPminer have always a good usage cohesion in the context of the validation client programs. Looking at the cohesion distribution in Figure 5, we observe that, for all studied APIs, the lower quartile in the case of COUPminer is higher than the upper quartile of NCBUPminer. This means that at least 75% of the cooperatively mined patterns are better than 75% of the patterns mined with NCBUPminer.

Patterns Consistency: As it can be seen in Table VI-B2, the results reveal that the patterns identified with NCBUPminer are overall characterized with very high consistency. Indeed, the average value across multiple validation client programs is around 94%. A slight deterioration is observed for the consistency of patterns mined with MLUPminer average values are around 81%. The table also shows that for all studied APIs, in the case of COUPminer, consistency values are close to those from NCBUPminer. We consider as an example the case of Swing API, where 22 leave-one-out cross validations have been made, using 22 client programs. In this case, the results show that, for any detected pattern in Swing, on average 87% (for COUPminer) and 94% (for MLUPminer) of co-usage

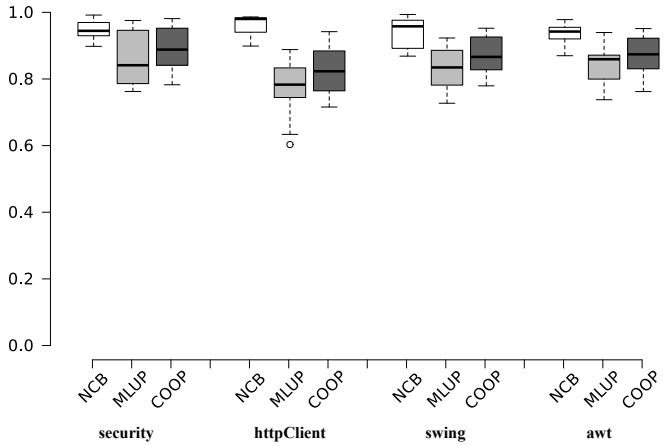

Figure 6. Consistency values of identified API usage patterns for NCBUPminer, MLUPminer and COUPminer across multiple validation clients.

relationships between pattern methods remain similar across 22 client programs of AWT. When comparing median boxplots in Figure 6 we note that, for all studied APIs, at least half of the patterns detected with COUPminer are more consistent than half of the patterns detected with MLUPminer.

C. Discussion and Threats to Validity

Overall, our technique COUPminer has been able to infer usage patterns that are more cohesive than those of NCBUPminer and more consistent than those of MLUPminer. In addition, and above all, as expected the cooperative cohesion was slightly lower than the one of MLUPminer, and the cooperative consistency was slightly lower than the one of NCBUPminer. However, the results are close and high enough to reflect accurate co-usage relationships and generalizable patterns. In conclusion, the cooperative approach allows achieving a very good tradeoff between the accuracy of the client-based mining and the generalizability of the library-based mining.

Detected usage patterns could help to enhance the APIs documentation and the evaluation of our approach took into account the generalization to other client programs of the same API and showed that the usage patterns of an API remain informative for new clients. As a threat to the validity of our study, this finding can be attributable to the client programs used in our study. Still, we used a fair number of validation client programs to evaluate the quality of the inferred usage patterns by our technique.

Another possible threat to validity, is the evaluation of our technique on four APIs only. Although, we selected four APIs of different domains, the approach may not be generalized to a larger set of APIs because these may be of different natures and quality. Indeed, we are combining, on the one hand, a library based heuristic that depends on the code quality and programming style, with, on the other hand, a client based heuristic that depends on the API size and the amount of functionalities provided to client programs. To better assess our cooperative approach, we are planning to investigate the effect of these properties on the effectiveness of our technique, in a larger experimental setting.

Table III
STATISTICS ON THE COHESION OF IDENTIFIED API USAGE PATTERNS FOR NCBUPMINER, MLUPMINER AND COUPMINER , IN THE CONTEXTS OF VALIDATION CLIENTS

	NCBUP			MLUP			COUP		
API	**Avg**	**StdDev**	**Max**	**Avg**	**StdDev**	**Max**	**Avg**	**StdDev**	**Max**
Security	0.70	0.04	0.76	0.85	0.12	1.00	0.86	0.12	1.00
HttpClient	0.69	0.07	0.78	0.79	0.16	1.00	0.79	0.08	1.00
Swing	0.60	0.04	0.68	0.85	0.09	1.00	0.80	0.06	1.00
AWT	0.68	0.06	0.77	0.84	0.06	0.95	0.81	0.05	0.88

Table IV
STATISTICS ON THE CONSISTENCY OF IDENTIFIED API USAGE PATTERNS FOR NCBUPMINER, MLUPMINER AND COUPMINER , ACROSS MULTIPLE VALIDATION CLIENTS.

	NCBUP			MLUP			COUP		
API	**Avg**	**StdDev**	**Max**	**Avg**	**StdDev**	**Max**	**Avg**	**StdDev**	**Max**
Security	0.94	0.03	0.98	0.86	0.09	0.98	0.88	0.07	0.98
HttpClient	0.96	0.03	0.97	0.77	0.09	0.89	0.83	0.08	0.94
Swing	0.94	0.05	0.98	0.83	0.05	0.92	0.87	0.05	0.95
AWT	0.92	0.03	0.89	0.84	0.04	0.94	0.86	0.07	0.95

Finally, one can question our adaptation of DBSCAN to produce multi-level usage patterns instead of using a classical hierarchical clustering technique. We opted for this solution since DBSCAN is known to be very robust to outliers.

VII. RELATED WORK

Helping developer to learn using an API gained a considerable attention in recent-year research. Several directions have been investigated, in particular, code search and completion [20]–[22], increasing awareness of the API change impact on their usability [23], [24], API usage pattern mining [5], [25]–[27], API specification mining [28]–[30], empirical study on API usability [31]–[36], API usability on question and answer (Q&A) systems [37], [38], API property visualization [39]–[41] and API documentation [42], [43]. For the sake of space, we limit our discussion to some of these contributions.

Zhong *et al.* [2] developed the MAPO tool for mining sequential API usage patterns. MAPO clusters frequent API method call sequences extracted from code snippets, based on the number of called API methods and textual similarity of class and method names between different snippets. Clustering only frequent call sequences could not be enough to detect less-common API usage scenarios.

Moritz *et al.* [44] present a technique for automatically mining and visualizing API usage examples that occur across several functions in a program. This approach represents software as a Relational Topic Model, where API calls and the functions that use them are modeled as a document network. The used model prevents finding common and consistent use, because no links can be represented between functions belonging to different client programs.

Wang *et al.* [5] use the BIDE [45] algorithm to mine frequent closed sequential patterns. The Authors opted for a two step clustering strategy, before and after BIDE, to identify patterns. The first clustering uses method call sequences as input , and the second one use frequent closed sequences as input. this technique aimed to reduce the number of redundant patterns and detects more unpopular patterns. In our case we are interested in detecting popular (i.e., generalizable) patterns.

Another body of works were based on automata mining to infer API properties [46]–[48]. Many techniques mine API specifications by encoding temporal order as finite-state automata. In our case and for scalability issue we decided not to mine usage patterns from complicated data representations such as automata.

Our approach tries to circumvent the limitations of the above-mentioned contributions. First, we make use of multiple clients of the studied API, which guarantees that usage patterns are genuine and recovered from the actual use of the API. Second we use similarity metrics within the library source code to reduce the problems of client program availability. Finally combining library-based and client-based techniques allows achieving both precision and generalizability of usage patterns.

VIII. CONCLUSION

We developed a technique that infers API usage patterns using both library-based and client-based heuristics. Our technique uses structural, semantic and co-usage similarity between the API methods to infer usage patterns. We applied our technique on four APIs that differ in size, utility and usage domains. To evaluate the performance of our technique, we analyzed the quality of inferred API patterns in terms of consistency and usage cohesion through a large variety of API client programs. We found that our technique allows taking advantage at the same time from the accuracy of the client-based techniques and the generalizability of the library-based techniques. As a future work, we plan to investigate other heuristics that may improve the quality of the patterns. We also believe that, we need to devote some research efforts to the visualization of API usage patterns.

REFERENCES

[1] G. Uddin, B. Dagenais, and M. P. Robillard, "Temporal analysis of api usage concepts," in *International Conf. on Software Engineering*, 2012, pp. 804–814.

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

[3] U. Dekel and J. D. Herbsleb, "Improving api documentation usability with knowledge pushing," in *International Conf. on Software Engineering*, 2009, pp. 320–330.

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

[5] 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 *Working Conf. on Mining Software Repositories*, 2013, pp. 319–328.

[6] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui, "Mining multi-level api usage patterns," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 23–32.

[7] Z. Zhu, Y. Zou, B. Xie, Y. Jin, Z. Lin, and L. Zhang, "Mining api usage examples from test code," in *International Conf. on Software Maintenance and Evolution*. IEEE, 2014, pp. 301–310.

[8] M. A. Saied, H. Abdeen, O. Benomar, and H. Sahraoui, "Could we infer unordered api usage patterns only using the library source code?" in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. IEEE Press, 2015, pp. 71–81.

[9] M. Ernst, "Static and dynamic analysis: Synergy and duality," in *ICSE Workshop on Dynamic Analysis*, 2003, pp. 25–28.

[10] "The jakarta commons httpclient component." [Online]. Available: http://hc.apache.org/httpclient-3.x/

[11] "The java.security package and all its subpackages (5 public packages in total)." [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/security/package-summary.html

[12] "The swing api (18 public packages in total)." [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/swing/

[13] "The java.awt package and all its subpackages (12 public packages in total)." [Online]. Available: http://docs.oracle.com/javase/7/docs/api/

[14] T. Xie and J. Pei, "Mapo: Mining api usages from open source repositories," in *International workshop on Mining software repositories*, 2006, pp. 54–57.

[15] 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.

[16] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge university press Cambridge, 2008, vol. 1.

[17] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, "Indexing by latent semantic analysis," *JASIS*, vol. 41, no. 6, pp. 391–407, 1990.

[18] M. Ester, H. peter Kriegel, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *International Conference on Knowledge Discovery and Data Mining*, 1996.

[19] M. Perepletchikov, C. Ryan, and K. Frampton, "Cohesion metrics for predicting maintainability of service-oriented software," in *International Conference on Quality Software*, 2007, pp. 328–335.

[20] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "Context-sensitive code completion tool for better api usability," in *International Conf. on Software Maintenance and Evolution*. IEEE, 2014, pp. 621–624.

[21] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding relevant functions and their usage," in *International Conf. on Software Engineering*, 2011, pp. 111–120.

[22] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2009, pp. 213–222.

[23] G. Bavota, M. Linares-Vásquez, C. Bernal-Cardenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "The impact of api change-and fault-proneness on the user ratings of android apps."

[24] W. Wu, B. Adams, Y.-G. Guéhéneuc, and G. Antoniol, "Acua: Api change and usage auditor," in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 2014, pp. 89–94.

[25] Y. Lamba, M. Khattar, and A. Sureka, "Pravaaha: Mining android applications for discovering api call usage patterns and trends," in *Proceedings of the 8th India Software Engineering Conference*. ACM, 2015.

[26] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy api usage patterns in android apps: an empirical study," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014.

[27] 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.

[28] D. Lo, S.-C. Khoo, and C. Liu, "Efficient mining of iterative patterns for software specification discovery," in *ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2007.

[29] D. Fahland, D. Lo, and S. Maoz, "Mining branching-time scenarios," in *International Conference on Automated Software Engineering (ASE)*. IEEE, 2013.

[30] D. Lo and S. Maoz, "Scenario-based and value-based specification mining: better together," *Automated Software Engineering*, vol. 19, no. 4, pp. 423–458, 2012.

[31] W. Wu, F. Khomh, B. Adams, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of api changes and usages based on apache and eclipse ecosystems," *Empirical Software Engineering*, 2015.

[32] L. Fischer and S. Hanenberg, "An empirical investigation of the effects of type systems and code completion on api usability using typescript and javascript in ms visual studio," in *Proceedings of the 11th Symposium on Dynamic Languages*. ACM, 2015.

[33] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom, "Use at your own risk: the java unsafe api in the wild," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2015, pp. 695–710.

[34] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to api deprecation?: the case of a smalltalk ecosystem," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 56.

[35] P. Petersen, S. Hanenberg, and R. Robbes, "An empirical comparison of static and dynamic type systems on api usage in the presence of an ide: Java vs. groovy with eclipse," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 212–222.

[36] S. Endrikat, S. Hanenberg, R. Robbes, and A. Stefik, "How do api documentation and static typing affect api usability?" in *International Conf. on Software Engineering*. ACM, 2014, pp. 632–642.

[37] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "How do api changes trigger stack overflow discussions? a study on the android sdk," in *International Conf. on Program Comprehension*. ACM, 2014, pp. 83–94.

[38] B. Zhou, X. Xia, D. Lo, C. Tian, and X. Wang, "Towards more accurate content categorization of api discussions," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014.

[39] R. G. Kula, C. De Roover, D. German, T. Ishio, and K. Inoue, "Visualizing the evolution of systems and their library dependencies," in *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*. IEEE, 2014, pp. 127–136.

[40] C. Parnin, C. Treude, L. Grammel, and M.-A. Storey, "Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow," *Georgia Institute of Technology, Tech. Rep*, 2012.

[41] M. A. Saied, O. Benomar, and H. Sahraoui, "Visualization based api usage patterns refining."

[42] M. A. Saied, H. Sahraoui, and B. Dufour, "An observational study on api usage constraints and their documentation," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 33–42.

[43] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 643–652.

[44] E. Moritz, M. Linares-Vasquez, D. Poshyvanyk, M. Grechanik, C. McMillan, and M. Gethers, "Export: Detecting and visualizing api usages in large source code repositories," in *Automated Software Engineering*, 2013, pp. 646–651.

[45] J. Wang and J. Han, "Bide: Efficient mining of frequent closed sequences," in *International Conf. on Data Engineering*, 2004, pp. 79–90.

[46] M. Acharya and T. Xie, "Mining api error-handling specifications from source code," in *Fundamental Approaches to Software Engineering*. Springer, 2009, pp. 370–384.

[47] D. Lo and S.-C. Khoo, "Smartic: towards building an accurate, robust and scalable specification miner," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2006, pp. 265–275.

[48] C. Lee, F. Chen, and G. Roşu, "Mining parametric specifications," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 591–600.