Could We Infer API Usage Patterns only using the Library Source Code?

Mohamed Aymen Saied, Hani Abdeen, Omar Benomar, and Houari Sahraoui
DIRO, Université de Montréal, Montréal, Canada
{saiedmoh,abdeenha,benomaro,sahraouh}@iro.umontreal.ca

***Abstract*—Learning to use existing or new software libraries is a difficult task for software developers, which would impede their productivity. Much existing work has provided different techniques to mine API usage patterns from client programs in order to help developers on understanding and using existing libraries. However, considering only client programs to identify API usage patterns is a strong constraint as the client programs source code is not always available or the clients themselves do not exist yet for newly released APIs. In this paper, we propose a technique for mining *Non Client-based Usage Patterns* (NCBUPminer). We detect API usage patterns as distinct groups of API methods that are structurally and semantically related and thus may contribute together to the implementation of a particular functionality for potential client programs. We evaluated our technique through four APIs. The obtained results are comparable to those of client-based approaches in terms of usage-patterns cohesion.**

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

I. Introduction

Software developers increasingly need to reuse functionality provided by external libraries and frameworks through their Application Programming Interfaces (APIs), where an API is the interface to implemented functionalities [1]. Hence, software developers have to cope with the complexity of existing APIs that are needed to accomplish developers' work.

Despite recent progress in API documentation and discovery [2]–[4], large APIs are difficult to learn and use [1], [3]. A large API may consist of several thousands public methods defined in hundreds classes. Since API classes are typically meant for a wide variety of usage contexts, the elaborated documentation of an API class may be very detailed as it tries to specify all aspects that a client might need to know about the class of interest. Hence, software developers might spend considerable time and effort to identify the subset of the class's methods that are necessary and sufficient to implement their daily work. Moreover, an API method is generally used within client programs along with other methods of the API, but it is not evident to deduce the co-usage relationships between API methods from their documentations. To overcome these difficulties in learning how to use API's methods, software developers may use source code search engines to search for usage examples. However, search engines usually return a huge number of code snippets that use the method of interest.

Therefore, identifying usage patterns for the API can help to better learn common ways to use the API, even if there exists several different ways to combine API elements (e.g., methods). In recent years, much research effort has been dedicated to the identification of API usage patterns [2], [3], [5], [6]. These existing techniques are valuable to facilitate API understanding and usage. Despite the different aspects they try to cover, these techniques are all based on client programs' code. However, client programs' code is unfortunately not available for both newly released API libraries and APIs which are not widely used. Even if client programs are available, all the usage contexts of the API of interest may be not covered by those clients. Indeed, from the coverage perspective, client-based identification of API usage patterns can be used only for a subset of the API of interest, that is the set of the API methods which are already used, multi-times, by different clients of the API. Hence, such techniques need to access and analyze the code of different clients of the API, and guaranty that those clients cover the possible variety of the API usage contexts.

In this paper, we propose a technique which does not rely on client programs' code to determine API usage patterns, namely Non Client Based Usage Patterns miner (NCBUPminer). Our approach is based on the idea that API methods can be grouped together based on 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. We are interested, in particular, in two types of relationships, namely, structural and semantic (conceptual) dependencies [7], [8].

NCBUPminer is premised on the analysis of structural and semantic dependencies of API methods within the API code itself. More precisely, we start from two assumptions: (1) the API methods that change the state and manipulate the same object could be complementary in their contribution to a functionality, i.e., structural relationship assumption; and (2) the API methods can be related to the same domain functionality if they share similar vocabulary, i.e., semantic relationship assumption.

The proposed approach is not an alternative to client-based ones. It is rather a solution when client programs are not available, i.e., for newly released API libraries and non-widely used ones. Therefore, we do not expect that it performs better for usage pattern identification. Still, our goal is to obtain results that are close to those of client-based approaches. In this context, to assess how much confidence we could have on the non-client based patterns, we performed a comparative evaluation of our technique NCBUPminer against a client based one MLUP [6]. We evaluated NCBUPminer using four different APIs: HttpClient [9], Java Security [10], Swing [11] and AWT [12]. To evaluate the derived patterns, we tested them on 12 client programs of the HttpClient and Java Security

APIs, and 22 client programs of the Swing and AWT APIs. The results show that across a considerable variability of client programs, usage patterns inferred by NCBUPminer remain sufficiently cohesive, as compared to those identified by MLUP. Before performing the above-mentioned comparison, we first evaluated the impact of the two used assumptions, structural and semantic dependencies, on the quality of inferred usage patterns, with regard to their actual co-usage relationships within client programs. The results show that, when structural relationships are used, the obtained patterns are more accurate. However, when both types of relationships are combined, the quality of the usage patterns is even better.

The remainder of the paper is organized as follows. Section II motivates the usefulness of this work with two actual examples from AWT and JavaSecurity APIs. We explain our approach in Section III and present the setting and API used for evaluating it in Section IV. Section V presents and analyzes the results of our study. In Section VI we discuss further our approach and the evaluation results. Finally, Section VII presents various contributions that are related to our work, before concluding in Section VIII.

II. Motivation Examples

In this section, we present two motivation examples to illustrate how the structural and semantic dependencies could be interpreted as an indicator of co-usage relationships.

A. Java Security Example

Java Security API [10] provides features to improve security of Java applications. The `KeyStore` class in the `java.security` API represents a storage facility for cryptographic keys and certificates. A `KeyStore` object manages different types of entries used to authenticate other parties such as `PrivateKeyEntry SecretKeyEntry` and `TrustedCertificateEntry`. Before a `Keystore` object can be accessed, it must be loaded, then, it would be possible to read/write entries from/into it.

NCBUPminer detected a usage pattern reflecting this functionality through the following 3 API methods:

1) `load(LoadStoreParameter)` is responsible to load `Keystore` objects using the given parameter.
2) `getEntry(String, ProtectionParameter)` gets the keystore entry for the specified alias with the specified protection parameter.
3) `setEntry(Str,Entry,ProtectionParameter)` saves the keystore entry under a specified alias and with respect to the protection parameter.

These methods have strong structural dependencies and semantic similarity. Precisely, both the `getEntry ()` and the `setEntry()` methods need to start by look up for the initialization state of the keystore through the *`initialized`* field in the `Keystore` class, which is set via the `load()` method. The aforementioned capabilities of the three methods are achieved via the *`KeyStoreSpi`* field, that defines the *Service Provider Interface* (SPI) for the `KeyStore` class. Moreover, `LoadStoreParameter` object, which must be passed to the `load()` method, is used to set the `ProtectionParameter` object, which is used to protect the keystore data. That object is then used as a parameter in both methods `getEntry()` and `setEntry()`.

B. AWT Example

AWT API [12] provides a reach toolkit for creating user interfaces and for painting graphics and images. We consider the class `Raster` in the AWT API. `Raster` is used to represent image data by a rectangular array of pixels. A `Raster` encapsulates a `DataBuffer` that stores the sample values of image bands datum and a `SampleModel` that describes the layout of the image data and how to locate a given sample value in a `DataBuffer`. Whenever a programmer needs to manipulate image low-level information, he can directly manipulate samples and pixels in the `DataBuffer` of the `Raster` class. For that, a conventional way consists on starting with the creation of a compatible sample model which describes the data of the manipulated image throw the method `createCompatibleSampleModel(int, int)` in the class `SampleModel`. In the second step, the programmer need to use the factory method `createWritableRaster(SampleModel, Point)` in the `Raster` class. This provides pixel writing capabilities throw the created `WritableRaster` object, and the programmer could manipulate the image low-level information. The next step would be the use of `createChild` method of the `Raster` class, to copy either all bands or only a subset of the image bounding rectangle. Finally the `setDataElements` method of the `WritableRaster` class is used to set the data for a rectangle of pixels from the manipulated image. Our technique detected these 4 methods as a usage pattern, since they are manipulating the same objects. For instance, returned objects by some methods of the pattern are used as parameters for the other methods. We can also notice the presence of some similarity between the vocabularies of these methods. Figure 1 shows a code snippet from the `SimpleRenderedImage` class in `GanttProject`, where the patterns method are used to copy a rectangular region.

III. Non Client Based Usage Patterns Analysis

This section presents our approach for detecting API usage patterns. Before detailing the used algorithm, we provide a brief overview of our approach and the representation of non-client based usage patterns in our approach.

A. Overview

We define a non-client based usage pattern for an API (an API usage pattern, UP, for short) as a subset of the API's methods that are structurally and semantically related. 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.

The rationale behind fetching the co-usage relationships in the API code itself is that public methods that change the state of or manipulate the same set of objects cooperate to accomplish certain domain functionality. However, even if

```
public WritableRaster copyData(WritableRaster dest) {
    Rectangle bounds;
    Raster tile;

    if (dest == null) {
        bounds = getBounds();
        Point p = new Point(minX, minY);
        /* A SampleModel to hold the entire image. */
        SampleModel sm = sampleModel.createCompatibleSampleModel(
                                    width, height);
        dest = Raster.createWritableRaster(sm, p);
                      ......

    for (int j = startY; j <= endY; j++) {
        for (int i = startX; i <= endX; i++) {
            tile = getTile(i, j);
            Rectangle intersectRect =
                bounds.intersection(tile.getBounds());
            Raster liveRaster = tile.createChild(
                    intersectRect.x,intersectRect.y,
                   intersectRect.width, intersectRect.height,
                   intersectRect.x,intersectRect.y,null);

            dest.setDataElements(0, 0, liveRaster);
        }
    }
    return dest;
}
```

Figure 1. Code snippets of `Raster` from `GanttProject`

some API methods are cooperating by manipulating the same object states, this cooperation could be for different domain purposes. Hence, those methods may not be co-used together, for one particular domain purpose, in client programs. As a matter of fact, the domain knowledge is encapsulated in the methods vocabulary [13], [14]. Therefore, our technique for identifying API usage patterns should isolate noises with respect to the degree of structural and semantics relationships in detected patterns.

Our approach takes as input the source code of the API to study and the output is a set of usage patterns as described earlier. The detection approach proceeds as follows.

- *Extracting API methods, references and terms.* First, the API source code is statically analyzed, and its public methods are retrieved. 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 those composing its parameter and the local variable identifiers.
- *Encoding methods information.* Then, we compute states and terms vectors for the API public methods. Each public method in the API is characterized by: (1) a vector of states 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 [15], [16] to construct a semantic space representation for the API of interest. The static analysis is performed using the Eclipse Java Development Tools (JDT).
- *Clustering.* Finally, we use cluster analysis to group the API methods that are most structurally and semantically related.

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 tow vectors. The first vector is the states' vector; it has constant length that is the number of all the manipulated classes and fields through the API public methods. Figure 2 shows an example where eight fields of four different classes are manipulated by the public methods of the API of interest. On that basis, the API methods will have a states' vector of length 12. For a given API method, an entry of 1 (or 0) in the i^{th} position of the states' vector, denotes that the i^{th} field is referenced (or not referenced) through the API method. If at least one field of a class is referenced, then the position of the corresponding class is also set to 1. This is done to, when computing the state similarity between two methods, give more importance to situations where both methods access fields from the same class than fields from different classes. The second vector is the terms' vector; it is computed from the lemmatized collected terms composing the public method name and its parameter and local variable identifiers. Similarly to states' vectors, the terms' vectors have constant length that is the number of all lemmatized collected terms composing the public methods in the API of interest. For a given API method, an entry of 1 (or 0) in the i^{th} position of the terms' vector, denotes that the i^{th} term appears (or does not appear) in the method vocabulary.

	C1	C2	C3	C4	C1.f1	C1.f2	C2.f1	C2.f2	C3.f1	C3.f2	C4.f1	C4.f2
API.m1	1	0	1	0	1	0	0	0	1	0	0	0
API.m2	1	0	1	0	0	1	0	0	0	1	0	0
API.m3	0	1	0	1	0	0	1	1	0	0	1	0
API.m4	0	1	0	1	0	0	1	0	0	0	1	1

Figure 2. The state vector representation of 4 API methods. In this API, 8 fields (`C1.f1 ... C4.f2`) of 4 different classes (`C1 ... C4`) are manipulated by the API methods.

The terms' vectors of the API are used in a Latent Semantic Indexing process [15] to create a term-document matrix C. Each of the rows of C represents a term, and each of its columns represents a document (i.e., an API public method). C is an $M \times N$ matrix where M is the number of all terms collected from the API and N is the number of public methods of the API. To better infer semantic similarity relations based on terms co-occurrence, a Singular Value Decomposition (SVD) [15] is applied. From the term-document matrix C, the SVD constructs three matrices: U_k is the SVD term matrix; Σ_k is the singular values' matrix; and V_k^T is the SVD document matrix, where in our case $k = min(M, N)$. The SVD document matrix, V_k^T, yields a new representation for each document (API public method), that enables us to compute document-document similarity scores in the semantic space representation as the cosine between the term vectors of the API methods.

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). For this purpose, we define two similarity metrics, State Manipulation Similarity $StateSim$ and

Semantic Similarity $SementicSim$.

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

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

As for the semantic similarity metric, $SementicSim$, 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 semantic information is shared among two API methods.

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

Where $\overrightarrow{V_i}$ and $\overrightarrow{V_j}$ are the j^{th} and i^{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.

Using the similarity metrics, we compute the distance between two points p_i and p_j representing respectively two API methods m_i and m_j as opposite to the average similarity between p_i and p_j:

$$Dist(P_i, P_j) = 1 - \frac{StateSim(m_i, m_j) + SementicSim(m_i, m_j)}{2} \quad (3)$$

D. Clustering algorithm

Our clustering is based on the algorithm DBSCAN [17]. DBSCAN is a density based algorithm, i.e., the clusters are formed by recognizing dense regions of points in the search space. The pseudocode of the DBSCAN algorithm is shown in Figure 3. 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 the clustering algorithm to leave out (not cluster) any point that is not located in a dense region of points in the search space. In other words, the algorithm clusters only relevant points and leaves out noisy points. This explains our choice of DBSCAN to detect API usage patterns.

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 to each other. In other words, epsilon value controls the minimal density that a clustered region can have. The shorter is the distance between the methods within a cluster the more dense is the cluster. The function `getNeighbors(P, epsilon)` in the pseudocode of the DBSCAN, in Figure 3, uses the distance defined in Equation (3) to decide if a point belongs to the neighborhood of a given point. In practice, the choice of the epsilon value is not straightforward, since we do not know before hand which density and which threshold of similarity between methods will lead to good-quality usage

```
DBSCAN(DataSet, epsilon, MinNbPts){
   clusters <- {}    //output of the algorithm
   noisyPoints <- {} //output of the algorithm
   for each unvisited point P in DataSet
      mark P as visited
      Neighborhood_P = getNeighbors(P, epsilon)
      if (Neighborhood_P.size) < MinNbPts
         noisyPoints <- noisyPoints + {P}
      else
         currentCluster <- new cluster
         constructCluster(P, Neighborhood_P,
               currentCluster, epsilon, MinNbPts)
         clusters <- clusters + {currentCluster}
}
constructCluster(P, Neighborhood_P, currentCluster, epsilon
       , MinNbPts){
   currentCluster <- currentCluster + {P}
   for each point Q in Neighborhood_P
      if Q is not visited
         mark Q as visited
         Neighborhood_Q <- getNeighbors(Q, epsilon)
         if Neighborhood_Q.size >= MinNbPts
            Neighborhood_P <- Union(Neighborhood_P,Neighborhood_Q)
      if Q is not yet member of any cluster
        currentCluster <- currentCluster + {Q}
}
getNeighbors(P, epsilon){
    for each point Q in DataSet
        if DIST(P,Q) < epsilon
        then Q is neighbor of P.
}
```

Figure 3. DBSCAN algorithm.

patterns. Therefore, as it will be shown in the next section, we will use different epsilon values for identifying our usage patterns. In other words, we adapt DBSCAN algorithm for identifying usage patterns that may have variant densities with regard to the similarity between the pattern methods. That is to avoid limiting our patterns by one, unjustified, threshold of similarity, which may lead to less-good solutions.

E. Incremental Clustering

In DBSCAN, the value of the epsilon parameter influences greatly the resulting clusters. Indeed, in our approach, a 0 value for epsilon means that each cluster must contain only API methods that are completely similar (i.e., distance among methods belonging to the same cluster must be 0). Relaxing the epsilon parameter will relax the constraint on the requested density within clusters.

.

In our approach, we decided to build the clusters incrementally by relaxing the epsilon parameter, step by step, as shows the pseudo-code of the incremental clustering in Figure 4. First, we construct a dataset containing all analyzed methods of the API of interest and cluster them using DBSCAN with an epsilon value of 0. This results in clusters of the most similar API methods, and multiple noisy points left out, i.e., points that could not be clustered because there is no other point exactly similar in terms of state and semantic similarity. Then, we construct for each produced cluster of this run a representative point (new state and term vectors), by aggregating the vectors of its composing methods 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:

$$\overrightarrow{T_k} = \Sigma_k^{-1} \times U_k^T \times \overrightarrow{T} \quad (4)$$

```
HDBSCAN(DataSet, maxEpsilon, MinPts, espilonStep){
  epsilon <- 0
  while(epsilon < maxEpsilon)
    DBSCAN(DataSet, epsilon, MinPts)
    clusters <- DBSCAN.clusters
    noisyPoints <- DBSCAN.noisyPoints
    compositePoints <- constructPoints(clusters)
    DataSet <- noisyPoints + compositePoints
    epsilon <- epsilon + epsilonStep
}

constructPoints(clusters){
  for each C in clusters
    stateVector <- OR(all stateVector of C)
    termVector <- OR(all termVector of C)
    lsiTermVector <- projectToLSI(termVector)
    compositePoint <-  compositePoint(stateVector,
           lsiTermVector)
}
```

Figure 4. Hierarchical DBSCAN algorithm.

A new dataset is formed including the new vectors and the noisy points from the first run. This dataset is fed back to the DBSCAN algorithm for clustering, but with a slightly higher value of epsilon. In the second run, some clusters form the first run are identical, other clusters are incremented with other points, and new clusters could be formed. The incremental clustering process is repeated until epsilon reaches a maximum value given as parameter.

IV. Evaluation

The objective of our study is to evaluate whether our technique can detect API usage patterns of good quality, that are comparable to those detected using several clients to the API of interest. We formulate the research questions of our study as follows:

- **RQ1:** *what is the quality of inferred usage patterns from the perspective of client programs?* and *what is the impact/contribution of the two used assumptions (structural and semantic)?*
- **RQ2:** *to which extent the inferred patterns are comparable to those detected by the client-based technique, MLUP?*

A. Comparative Evaluation

To fairly evaluate the quality of our detected API usage patterns from the perspective of the API clients, and compare them to client-based detected usage patterns, we evaluate our technique using four widely known APIs: HttpClient, Java Security, Swing and AWT (Table I). To perform our study, we selected 22 client programs for the Swing and AWT APIs, and 12 client programs for the HttpClient and Java Security APIs (see Table II).

To address our first research question, **RQ1**, we apply our technique on the four selected APIs and analyze the quality of detected patterns in the contexts of selected client programs, using the parameters and metrics that we detail in Section IV-B. To analyze the impact of different used assumptions (*state similarity* vs. *semantic similarity*) on our detected usage patterns, we use each heuristic alone for detecting patterns in the selected APIs, and compare the quality of detected usage patterns. We also evaluate the quality of the patterns produced by considering the combination of both assumptions.

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 our technique for mining Non Client-based Usage Patterns (NCUPminer) to the most similar client-based approach (MLUP) [6]. To mine API usage patterns, MLUP applies also the DBSCAN algorithm to cluster API methods analyzing the frequency of co-usage relations between the APIs methods within a variety of client programs of the API. We compare with MLUP since it is a client-based approach, and both approaches, MLUP and NCUPminer, use the same clustering algorithm. For a fair comparison, we need to assess how the newly proposed heuristics in this work (structural and semantic similarity) will perform on the same set of API methods that can be clustered using MLUP, i.e., only the subset of methods used by the considered clients.

B. Experimental Setup

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

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 enough cohesive 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), that was previously adopted in [6] to evaluate the quality of API usage patterns detected with MLUP. PUC was originally proposed and used in [18], [19] for assessing the usage cohesion of interfaces. 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. The PUC for a given usage pattern *p* is defined as follows:

$$PUC(p) = \frac{\sum_{cm} ratio\_used\_mtds(p, cm)}{|CM(p)|} \quad (5)$$

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, c)* 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*.

To assess the quality of inferred API usage patterns using NCBUPminer, and analyze the impact of different used

Table II
CLIENT PROGRAMS USED IN OUR CASE-STUDY

APIs	Client programs	Description
Swing & AWT	G4P (GUI for processing)	A library that provides a rich collection of 2D GUI controls
	Valkyrie RCP	A Spring port of the current Spring Rich Client codebase.
	GLIPS Graffiti editor	A cross-platform SVG graphics editor
	Mogwai Java Tools	Java 2D and 3D visual entity relationship design and modeling (ERD,SQL)
	Violet	An UML editor
	Mobile Atlas Creator	This application creates off-line raster maps
	Metawidget	A smart widget Building User Interfaces for domain objects
	Art-of-Illusion	A 3D modelling and rendering studio
	VASSAL	An engine for building and playing human-vs-human games
	Neuroph	A lightweight Java Neural Network Framework
	WoPeD	A Java-based graphical workflow process editor, simulator and analyzer
	jEdit	A mature programmer's text editor
	Spring-RCP	Provide a way to build highly-configurable, GUI-standards
	GanttProject core	An application for project management and scheduling
	Pert	The PERT plugin for GanttProject
	Htmlpdf	The html and pdf export plugin for GanttProject
	Msproject	MS-Project import/export plugin for GanttProject
	swingx	Contains extensions to the Swing GUI toolkit
	JHotDraw	A Java GUI framework for technical and structured Graphics
	RapidMiner	An integrated environment for machine learning and data mining
	Sweet Home 3D	An interior design application
	LaTeXDraw	Is a graphical drawing editor for LaTeX
HttpClient & Java Security	Jakarta Cactus	A simple test framework for unit testing server-side java code
	Apache JMeter	A project that can be used as a load testing and measure performance tool
	Heritrix	A web crawler
	HtmlUnit	A GUI-Less browser for Java programs
	OpenLaszlo	An open source platform for the development and delivery of rich Internet applications
	Mule	A lightweight enterprise service bus (ESB) and integration framework
	RSSOwl	An aggregator for RSS and Atom News feeds.
	Apache Jackrabbit	Is an open source content repository for the Java platform.
	Axis2	A core engine for Web services.
	RESTEasy	A JBoss project that provides various frameworks to build RESTful Web Services
	WildFly	An application server
	WSO2 Carbon	An SOA middleware platform

heuristics (**RQ1**), we run NCBUPminer three times on each studied API. Each run uses all the API's public methods as the data set to be clustered. In the first run, we consider both heuristics, structural and semantic similarity between methods, whilst in the second and third runs, we dissociate the two heuristics and consider just structural similarity and semantic similarity, respectively. For each studied API, we collect the inferred API usage patterns for the three runs and analyze their quality w.r.t. their usage cohesion (i.e., PUC values) in the context of the API client programs in Table II. Note that some of the inferred patterns by our technique, NCBUPminer, may not be covered by the selected client programs –although we use a large variety of client programs for each studied API. 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. In addition to the usage cohesion property, we also compare the number and average size of inferred patterns in each run. Based on the comparison results, we decide what heuristic/s is/are the best for inferring API usage patterns using NCBUPminer.

To answer our second research question **(RQ2)**, for all the selected APIs we apply NCBUPminer (using both heuristics) and MLUP (using the APIs client programs described in Table II). For a given API, only the API's methods which are covered by the analyzed client programs of the API can be clustered by MLUP. Hence, for a fair comparison between NCBUPminer and MLUP, first we identify the set of methods that can be clustered by MLUP, then we use only this set of methods as an input for NCBUPminer and MLUP. Then, we compare the detected usage patterns through the two techniques, w.r.t. their usage cohesion in the context of the API's client programs, that MLUP used for identifying its usage patterns. We collect the PUC values of identified patterns by NCBUPminer and MLUP, and use the Wilcoxon rank sum test with a 95% confidence level to test whether a significant difference exists between the measurements for the two techniques. Moreover, we opt for Cliff's delta tests for estimating the effect-size delta between the cohesion scores of usage patterns identified by the two techniques.

As both techniques use the DBSCAN incremental clustering algorithm, we set the maximum epsilon value for both techniques, in all runs, to 0.35. In our approach, this value can be interpreted as follows: the value of maximal

(structural/semantic) distance between two methods within an inferred pattern should be smaller than 0.35 (where 0 and 1 are, respectively, the smallest and largest values of distance). As for MLUP, this value means: for all detected patterns, the maximal non-uniformity in the co-usage of the pattern's methods should be smaller than 0.35.

V. Results Analysis

In the following paragraphs, we report the results of our experiments.

A. Impact of Used Heuristics (RQ1)

To answer our research question RQ1, we analyze the quality (usage cohesion) of inferred API usage patterns, as well as their number and size, and we inspect the impact/contribution of different used heuristics.

Usage Cohesion: As illustrated in Figure 5, for all the studied APIs, when only the semantic similarity between API methods is used, the inferred usage patterns reflect less-good co-usage relationships between the pattern's methods. Indeed, the average usage cohesion values for this case are between 44% for HttpClient and 52% for Java Security. Nevertheless, when the structural similarity between the API methods is considered, the usage cohesion values of inferred usage patterns are improved. Using this heuristic alone, on average 61% and up to 69% of the pattern's methods are uniformly co-used together. The obtained cohesion values using this heuristic alone are overall acceptable. However, they are still lower than the average usage cohesion value obtained for the Java Security API while combining both heuristics, which is 73%. In this last case, we notice that the average usage cohesion is high, but with a slight degradation in the case of Swing API, where the average usage cohesion is 64%. This is mainly due to the large size of Swing. Indeed, Swing declares 7226 public methods, as compared to Java Security API which declares 901 public methods. Despite the large number of declared public methods in Swing, and with regard to 22 different clients of this API, the results show that for an inferred usage pattern by NCBUPminer on average 64% of the pattern's methods are uniformly co-used together.

Number of Inferred Patterns: Figure 6 shows that an order relation can be observed between the numbers of patterns inferred through the three heuristics. For all studied APIs, the lowest number of inferred usage patterns was obtained while only the semantic heuristic is used. In second place, while combining the two heuristics the number of inferred patterns for each studied API was much greater than the previous case. Here, the number of inferred patterns for the Swing API reached a peak of 275 patterns.

Size of Inferred Patterns: The results in Figure 7 show that either when only the structural heuristic is used or both the structural and semantic heuristics are combined, the obtained sizes are almost equivalent with more or less three methods per usage pattern. The largest usage patterns were inferred while only the semantic heuristic is used. Here, we observe that the average sizes of inferred clusters for HttpClient, Swing and AWT are around 15, 17 and 22, respectively. This can explain the small number of inferred patterns using this heuristic, as

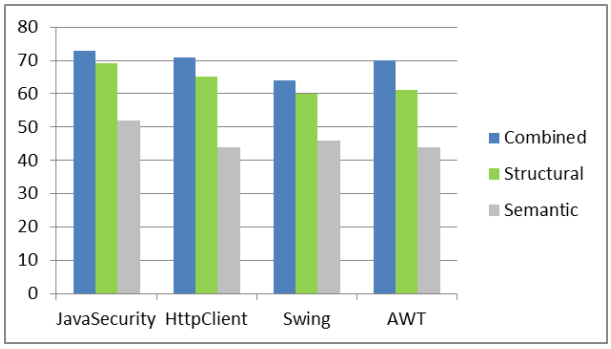

Figure 5. Average *cohesion* of inferred patterns using different heuristics

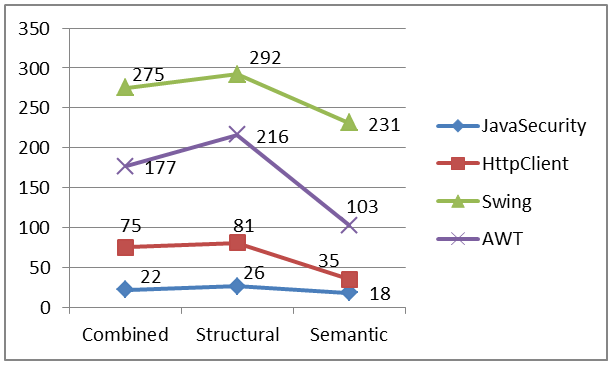

Figure 6. Average *number* of inferred patterns using different heuristics

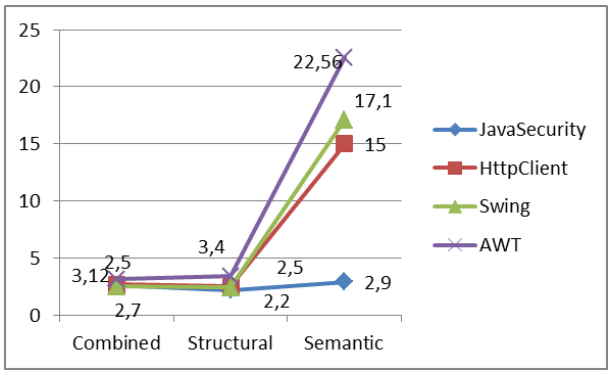

Figure 7. Average *size* of inferred patterns using different heuristics

well as the low usage cohesion of the inferred patterns. When analyzing the inferred patterns using this heuristic, we found that they group methods having similar vocabulary but in most cases contribute to different functions in the domain. For some inferred clusters, we found that clustered methods belong in general to different classes implementing the same interface. Moreover, we were able to identify sub-clusters of methods that have strong structural similarity and contribute to one specific function in that vocabulary domain.

As a summary, the results show that NCBUPminer can infer co-usage relationships between the API's methods with high precision. And, the structural similarity between the API methods has better contribution than the semantic similarity for inferring good API usage patterns. Still that, combining both heuristics NCBUPminer performs the best for inferring the co-usage relationships. In this case, with regard to all

studied APIs and their analyzed client programs, on average 64% and upto 73% of methods in an inferred usage pattern are always uniformly co-used together, that is in the context of a large variety of API client programs. More specifically, using the structural similarity leads to cluster the API's methods which tightly collaborate together, but might contribute to different domain functions. The semantic heuristic enables our technique to identify clusters of methods that belong to the same vocabulary domain, and the structural heuristic enables it to identify sub-groups in those clusters, in which methods contribute together to one specific functionality in that domain.

Such good usage patterns detected by NCBUPminer, while combining both heuristics, are the examples that we outlined in Section II. In addition to those patterns, NCBUPminer was able to infer other informative API usage patterns that were not covered/used by the API client programs considered in our study –although we used a large variety of client programs for each studied API. For instance, NCBUPminer inferred a usage pattern of the HttpClient API for validating certificate chain. The inferred pattern consists of the following four methods defined in the classes `PKIXParameters` (methods 1 and 2) and `CertPathValidator` (methods 3 and 4):

1) `PKIXParameters(java.security.KeyStore)`,
2) `setRevocationEnabled(boolean)`,
3) `getInstance(String)`, and
4) `validate(CertPath, CertPathParameters)`

Although these methods are not covered by the 12 client programs of HttpClient API that we used in our study, our analysis revealed that they form an informative usage pattern of HttpClient API. Indeed, using code search engines, we found that these methods are uniformly used together in several client programs of HttpClient API, as in the `Waterken`[1] project, for the purpose of validating certificate chains. For instance, the code snippet in Figure 8 shows how this pattern is used in a method of the URL `Handler` class in `Waterken`, for checking the trusted server. The description of this usage pattern is as follows. First the method `PKIXParameters()` is used to create an instance of `PKIXParameters`, that the `CertPathValidator` will use to validate certification chains. Then, the second method `setRevocationEnabled()` is used to enable or disabled the default revocation checking mechanism of the underlying PKIX (Public-Key Infrastructure X.509) service provider. In the last step, the method `getInstance()` is invoked to create a `CertPathValidator` object which implements the specified algorithm that can be used to validate certification paths by calling the fourth method `validate()`.

B. Comparative Evaluation (RQ2)

To address our research question **RQ2**, we applied our technique (with the combined assumptions) and MLUP for detecting usage patterns of selected APIs.Then, we compared the results.

We, first, analyze the average usage cohesion values for all detected usage patterns per studied API obtained after using the two techniques as shown in Table III. The results reveal that using both NCBUPminer and MLUP, the identified usage patterns for the four studied APIs are overall characterized with high usage cohesion values, which reflect very strong co-usage relationships between the methods of identified patterns. Indeed, the average usage cohesion values of identified patterns across multiple validation client programs are around 80% for NCBUPminer and 90% for MLUP. Although the results of NCBUPminer are, as expected, slightly lower than the one of MLIP, they are close and higher enough, considering that the client programs are not seen in the derivation process. Actually, using the Wilcoxon rank sum test, we found that the deference between the two compared approaches, with regard to the usage cohesion of detected patterns, is not statistically significant at $\alpha = 0.05$. To estimate the effect-size between the usage cohesion scores of NCBUPminer patterns and that of MLUP patterns, we performed Cliff's delta test at 95% confidence interval. The result is that the estimated delta value, which is $d = 0.10$ in favor of MLUP patterns, is not significant. More precisely, there is a probability of only 42.8% that a pattern randomly chosen from MLUP results will have a higher usage cohesion score than a randomly chosen pattern from NCBUPminer patterns, as compared to 32.3% probability in favor of NCBUPminer, and to 25% probability for the equality of scores. Hence, we state that the performance of NCBUPminer for inferring API usage patterns is comparable to that of MLUP.

Table III
AVERAGE COHESION OF IDENTIFIED API USAGE PATTERNS, FOR NCBUPMINER AND MLUP

API	NCBUPminer	MLUP
Java Security	0.84	0.90
HttpClient	0.83	0.96
Swing	0.78	0.94
AWT	0.81	0.94

Table IV summarizes the number and size of API usage patterns identified by NCBUPminer and MLUP in this phase. The table shows that, for all studied APIs, MLUP has been able to identify more usage patterns than NCBUPminer. Moreover, MLUP usage patterns are, overall, slightly larger than NCBUPminer ones. We believe that this is mainly due to the capabilities of MLUP in resolving the interference in co-usage relationships between API methods. Indeed, thanks to the valuable information about the usage of API methods in the API client programs, MLUP can identify API's methods that do not belong to the same vocabulary domain, but their usage frequently interferes with each others. Still, except for Java Security, the number of patterns inferred by NCBUPminer is comparable to one of MLUP. In HttpClient, Swing and AWT, the percentages are, respectively, 95% ($^{19}/_{20}$), 91% ($^{93}/_{102}$), and 77% ($^{58}/_{75}$).

In conclusion, the detected usage patterns with NCBUPminer retain their informative criteria independently of the API usage scenarios, and our technique can be used to enhance the API documentation with co-usage relationships with high confidence, when the client programs are not available or are

[1] Waterken is a platform for secure interoperation using a capability messaging protocol.

```
public void checkServerTrusted(X509Certificate[] chain, String authType) throws CertificateException{
                // Validate the certificate chain.
                                        ...
                    PKIXParameter params = new PKIXParameters(Collections.singleton(ta));
                    params.setRevocationEnabled(false);
                    CertPathValidator.getInstance("PKIX").validate(path, params);
                                        ...
    }
```

Figure 8. Code snippet for validating certificate Chain, in method `checkServerTrusted` from class `Handler` in `Waterken`.

Table IV
OVERVIEW ON THE NUMBER OF COVERED/ANALYZED METHODS AND THE NUMBER OF DETECTED USAGE PATTERNS PER API

		NCBUPminer				MLUP			
			UP's size				UP's size		
API	**cov. mtd**	**UPs**	**Avg**	**Min**	**Max**	**UPs**	**Avg**	**Min**	**Max**
Java Security	125	4	2.5	2	3	12	2.8	2	4
HttpClient	343	19	2.4	2	4	20	2.7	2	4
Swing	1618	93	2.8	2	8	102	3.9	2	7
AWT	1019	58	2.5	2	6	75	4.3	2	9

not numerous enough to cover the multiple usage scenarios. Indeed, the majority of usage patterns detected by the client-based technique, MLUP, were inferred by our non-client based technique, NCBUPminer.

VI. DISCUSSION

Software clustering techniques, which are based on structural or semantic coupling between software entities, have been widely used to support program comprehension, software remodularization, concept location or feature identification [7], [8]. However, this is the first time it is shown that combining both heuristics, structural and semantic similarity between methods, can lead to identifying new dimensions of dependencies between API methods, that are co-usage relationships within API client programs. In this work, we evaluated the impact of the two aforementioned heuristics on the quality of inferred API usage patterns. The results show that combining both heuristics performs the best for inferring the co-usage relationships between the API's methods within client programs.

The valuable contribution of our technique over existing techniques around identifying API usage patterns, is that it can be applied on "new" APIs, where client programs are not available. In fact, our technique can be used even before the release of the API for assisting API developers in comprehension tasks and in enriching the API documentation.

To evaluate the performance of our technique in inferring API usage patterns, we compared it to the most similar client-based approach MLUP [6]. We compared with MLUP since both techniques use the same clustering algorithm, and the patterns detected with MLUP have the valuable property of generalizability and could reflect interfering API usage scenarios. As both techniques use the DBSCAN incremental clustering algorithm, we used for both the same configuration used in MLUP in [6], where the value of the parameter *maxEpsilon* is set to 0.35. However, as the comparative techniques use different heuristics, the parameter *maxEpsilon* has a completely different interpretation in each technique.

We applied our approach to four APIs and detected usage patterns that are informative and, which could help to enhance the APIs documentation. We state that the performance of our technique for inferring API usage patterns is comparable to that of MLUP. However, as a threat to validity, this finding is related to the used client programs in our study. Still, our study used a fair set of validation client programs to evaluate the quality of inferred usage patterns by our technique.

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], [21], [22], [23], [24]), increasing awareness of the API change impact on their usability [25]–[27], API usage pattern mining ([28], [5]) and API documentation ([29], [4], [30], [31]). For the sake of space, we limit our discussion to some of these contributions.

Uddin *et al.* [2] detect temporal patterns of API use in terms of their time of addition/removal into the source code during the client program development. Due to the volatile nature of the development process detected patterns may suffer from some imprecision.

Zhong *et al.* [3] developed the MAPO tool for mining 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. Moreover, textual similarity can be a barrier, since it is very common that snippets with different usage contexts refer to the same usage pattern.

Moritz *et al.* [32] 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. In the same spirit of our approach, namely independence from client programs, Zhu *et al.* [33] propose an approach to mining API usage examples from unit test code. They employed a slicing based approach to separate test scenarios into code examples. Then they clustered the similar usage examples for recommendation. Although their approach overcomes the difficulties related to redundant client program availability, the issue of usage representative test scenarios might lead to atomic examples. Moreover, as for client programs, test code is not always available for all

APIs, and if available, it usually does not cover the whole API functionality.

Our approach tries to circumvent the limitations of the above-mentioned contributions. First, we propose to infer API usage patterns only using the library source code which address the problems of client program or test code availability. Second, combining the structural and semantic similarity between the APIs methods performs the best for inferring co-usage relationships independently from usage context. Finally, detected usage patterns with our approach retain their informative criteria independently of the API usage scenarios.

VIII. Conclusion

We developed a technique that infers API usage patterns using only the API source code, independently of the availability of API client programs or unit tests to cover the API functionality. Our technique uses only structural and semantic similarity between the API methods to infer their co-usage relationships. 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 usage cohesion using a large variety of API client programs. We found that our technique can infer API usage patterns with a precision that is comparable to the most-recent client-based technique for inferring API usage patterns. Furthermore, we evaluated the contribution of each used heuristic for inferring good usage patterns. We found that by combining both heuristics, structural and semantic similarity, our technique performs the best. As a future work, we plan to investigate other heuristics that may improve the quality of the patterns. We believe that, to be pragmatic, more research effort needs to be devoted to the non-client based mining of usage patterns as very few libraries can have publicly accessible client programs.

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

[3] 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.

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

[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] S. Mohamed Aymen, B. Omar, A. Hani, and S. Houari, "Mining multi-level api usage patterns," in *International Conf. on Software Analysis, Evolution, and Reengineering*. IEEE, 2015. [Online]. Available: http://www-etud.iro.umontreal.ca/~saiedmoh/papers/SANER2015-Multi-levelAPIUsagePatterns.pdf

[7] G. Scanniello and A. Marcus, "Clustering support for static concept location in source code," in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, 2011, pp. 1–10.

[8] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. d. Lucia, "Improving software modularization via automated analysis of latent topics and dependencies," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 1, pp. 4:1–4:33, Feb. 2014.

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

[10] "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

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

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

[13] L. Guerrouj, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An experimental investigation on the effects of context on source code identifiers splitting and expansion," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1706–1753, 2014.

[14] V. Arnaoudova, L. Eshkevari, M. Di Penta, R. Oliveto, G. Antoniol, and Y. Gueheneuc, "Repent: Analyzing the nature of identifier renamings," 2014.

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

[16] 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.

[17] 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, pp. 226–231.

[18] 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.

[19] H. Abdeen, H. Sahraoui, and O. Shata, "How we design interfaces, and how to assess it," in *International Conference on Software Maintenance*, 2013, pp. 80–89.

[20] E. Duala-Ekoko and M. P. Robillard, "Using structure-based recommendations to facilitate discoverability in apis," in *European Conf. on Object-oriented Programming*, 2011, pp. 79–104.

[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] 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.

[24] 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.

[25] 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."

[26] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "Api change and fault proneness: a threat to the success of android apps," in *Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 477–487.

[27] 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.

[28] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining api patterns as partial orders from source code: From usage scenarios to specifications," in *Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2007, pp. 25–34.

[29] R. P. L. Buse and W. Weimer, "Synthesizing api usage examples," in *International Conf. on Software Engineering*, 2012, pp. 782–792.

[30] S. Mohamed Aymen, S. Houari, and D. Bruno, "An observational study on api usage constraints and their documentation," in *International Conf. on Software Analysis, Evolution, and Reengineering*. IEEE, 2015.

[31] J. Montandon, H. Borges, D. Felix, and M. Valente, "Documenting apis with examples: Lessons learned with the apiminer platform," in *Working Conf. on Reverse Engineering*, 2013, pp. 401–408.

[32] 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.

[33] 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.