An Empirical Study on API Usages

Hao Zhong, *Member, IEEE,* and Hong Mei, *Fellow, IEEE*

Abstract—API libraries provide thousands of APIs, and are essential in daily programming tasks. To understand their usages, it has long been a hot research topic to mine specifications that formally define legal usages for APIs. Furthermore, researchers are working on many other research topics on APIs. Although the research on APIs is intensively studied, many fundamental questions on APIs are still open. For example, the answers to open questions, such as which format can naturally define API usages and in which case, are still largely unknown. We notice that many such open questions are not concerned with concrete usages of specific APIs, but usages that describe how to use different types of APIs. To explore these questions, in this paper, we conduct an empirical study on API usages, with an emphasis on how different types of APIs are used. Our empirical results lead to nine findings on API usages. For example, we find that single-type usages are mostly strict orders, but multi-type usages are more complicated since they include both strict orders and partial orders. Based on these findings, for the research on APIs, we provide our suggestions on the four key aspects such as the challenges, the importance of different API elements, usage patterns, and pitfalls in designing evaluations. Furthermore, we interpret our findings, and present our insights on data sources, extraction techniques, mining techniques, and formats of specifications for the research of mining specifications.

Index Terms—API usage, mining specification, empirical study.

1 Introduction

As a successful example of information hiding [54], Application Programming Interfaces (APIs) are widely used in the modern software industry. Although reusing APIs reduces programming effort, researchers and practitioners complain that APIs are often poorly documented [61] and difficult to use [46]. After decades of development, software repositories accumulate many source files that illustrate API usages. In recent years, it has been a hot research topic to mine specifications from such source files, and a mined specification defines the legal sequences or the invariants (*e.g.*, preconditions) for calling APIs. Robillard *et al.* [60] present a comprehensive survey on this research direction. These approaches mine many specifications in various formats (*e.g.*, frequent call sequences [90], automata [5], temporal logics [76], and graphs [50]), and researchers have applied such mined specifications in detecting bugs [29], monitoring anomaly behaviors [19], and recommending code samples [90]. Besides mining specifications, researchers have proposed other approaches that assist programming with APIs more effectively. For example, researchers have proposed approaches that update API calls from an obsolete version to a recent version [10], [81] and migrate API calls across languages [47], [89].

Although the research on APIs is intensively studied, many fundamental questions are still not fully explored. For example, in literature, researchers proposed approaches that mine two types of specifications: single-type specifications that define API usages of individual API classes (*e.g.*, [92]), and multiple-type specifications that define API usages of multiple API classes (*e.g.*, [90]). However, the two types of approaches are never compared with each other, and researchers even do not know which specifications fit their needs. As another example, researchers have proposed approaches that update API calls from an obsolete version to a recent version. In their evaluations, most approaches (*e.g.*, [10]) present their effectiveness in updating individual libraries. Until now, researchers do not know whether multiple libraries are typically used in the same piece of code. If they are, evaluating on individual libraries is insufficient to prove the effectiveness of proposed approaches. In total, our study explored the following six research questions:

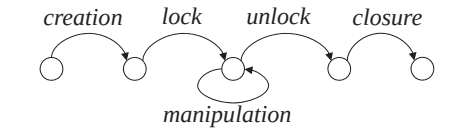
- What is the role of API fields and static API elements?
- Which is the best format to define API usages?
- What is the right tradeoff between single-type usages and multiple-type usages?
- To what degree do programmers work with APIs from different libraries?
- What are the proper lengths for API usages?
- How frequently are APIs used?

Section 3.1 presents the details of the above research questions, and Section 3.2 analyzes the significance of these research questions, as far as mining specification is concerned.

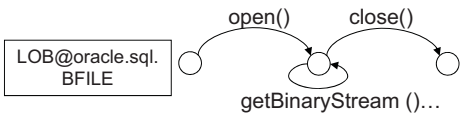
Benefits. It is desirable to answer the above research questions, and the benefits are as follows:
Benefit 1. With the answers to the above questions, researchers can make better choices, when they design their approaches or evaluations in future work. In addition, researchers can also revisit and tune their proposed approaches, according to the answers.
Benefit 2. With the answers to the above questions, researchers can rethink whether the answers are consistent with their intuitions. Indeed, an inconsistency can indicate a neglected problem, which may need further exploration. For example, many approaches (*e.g.*, [90]) mine multiple-type specifications as sequences. As Finding 6 shows that graphs are more suitable to encode multiple-type usages, researchers can extend their approaches with graphs.

As APIs are many and complicated, even experienced programmers may not fully understand their usages. To deepen the knowledge on APIs, researchers have conducted empirical studies (see Section 7 for details), but prior studies do not touch our open questions. In addition, some empirical studies (*e.g.*, [66]) were manually analyzed, which can be biased and does not scale. To answer the open questions, it is desirable to automate the process.

- *H. Zhong and H. Mei are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China, 200240. E-mail: {zhonghao,meih}@sjtu.edu.cn*



(a) The specification template



(b) A sample specification

Fig. 1. The bias in Zhong *et al.* [92]

Challenges. Intuitively, as specifications formally define API usages, analyzing mined specifications can provide valuable hints to the open questions. However, it is difficult to answer the open questions in this way, due to the following challenges:
Challenge 1. It is difficult to analyze mined specifications, since they are in various formats and are difficult to obtain. Furthermore, Robillard *et al.* [60] show that many mined specifications are false. It can be nontrivial to identify correct specifications.
Challenge 2. It is insufficient and biased to analyze mined specifications. As discussed before, researchers often rely on their personal experience to design their approaches. The limitation or bias in their personal experience can lead to bias in analysis results. For example, Zhong *et al.* [92] rely on a template to mine specifications for resources (*e.g.*, files and database connections). Figure 1(a) shows the template, and it defines the typical actions such as creation, lock, manipulation, unlock, and closure. For a resource, Zhong *et al.* [92] link its methods to corresponding actions, when they infer its specification. For example, Figure 1(b) shows an inferred specification for the `LOB` resource. If we analyze specifications mined by Zhong *et al.* [92], we can obtain only the knowledge on resources, since all their specifications are inferred from a resource template.

Our insight. Instead of analyzing mined specifications, we extract API usages directly from client code. As we do not analyze the mined specifications of specific approaches, we eliminate the bias inside selected mining approaches. Furthermore, we notice that the open questions in Section 3.1 are not concerned with usages of specific concrete APIs, but how to use different types of APIs. For example, if we divide API usages into single-type usages and multiple-type usages, it becomes feasible to answer which is more common and in which cases.

Our contribution. In this paper, we implemented a tool that analyzes how programmers use different types of APIs. With its support, we conducted an empirical study on API usages. Our study leads to the following results:

- **The importance and the challenges of the research on APIs.** Researchers have proposed many approaches that mine specifications from client code. Despite of their positive results, we find that most APIs do not have much client code, so we have to learn their usages from other data sources such as documents (Finding 9).
- **API elements.** It is reasonable for researchers to focus on methods, since methods are more frequently called than fields (Finding 2). While methods appear in both strict orders and partial orders, fields are more frequently used in partial orders (Finding 3). Researchers can pay more attention to static methods, since they are frequently called and their usages are different (Finding 1).
- **Usage patterns.** Strict orders and partial orders roughly divide API usages into two halves (Finding 4). Single-type usages and multi-type usages are equivalently common (Finding 5). Most single-type usages are strict orders, but multi-type usages include as many strict orders as partial orders (Finding 6). API usages typically are short but with some quite long outliers, and partial orders are often longer than strict orders (Finding 8).
- **Evaluation.** Most usages call no more than two libraries, and when more than one library is called, one of them is typically J2SE (Finding 7). As a result, although evaluating individual libraries can lose many usages, it is sufficient to add J2SE to individual libraries, when researchers design evaluations.

More specifically, in Section 6, we present the interpretation of our findings, as far as mining specifications is concerned. The other sections of this paper are organized as follows. Section 2 introduces the background of our study. Section 3 introduces our research goal. Section 4 presents our analysis methodology. Section 5 presents our empirical study. Section 7 presents related work. Section 8 concludes and discusses the future work.

2 Mining Specification

As introduced in Section 1, many papers on APIs are related to mining specifications. In this section, for the research in mining specifications, we briefly introduce its data sources (Section 2.1), techniques (Section 2.2), and mined specifications (Section 2.3).

2.1 Data Source

Definition 1. An Application Programming Interface (API) is a set of visible code elements provided by frameworks or libraries, and such frameworks or libraries are called API libraries. The code of API library is called API code.

For most commercial libraries, API code is not available. Most approaches consider API code as a black box, with only several exceptions (*e.g.*, [12], [91]).

Definition 2. Client code is application code that reuses or extends code elements provided by API libraries.

The definitions of API code and client code are relative to each other. For example, Lucene uses classes and methods provided by J2SE[1], so we consider Lucene as client code and J2SE as API code, when we analyze the code of Lucene. Meanwhile, Nutch[2] uses classes and methods provided by Lucene, so we consider Nutch as client code and Lucene as API code, when we analyze the code of Nutch. The following code further illustrates the concept:

```
1: import java.io.BufferedReader;
2: import java.io.BufferedWriter;
3: import java.io.InputStreamReader;
4: import java.io.OutputStreamWriter;
5: import java.io.PrintWriter;
6: import java.net.ServerSocket;
7: import java.net.Socket;...
8: public static void main(...){
9:   ...
10:  ServerSocket s = new ServerSocket(8080);
11:  Socket socket = s.accept();
12:  BufferedReader br = new BufferedReader(
       new InputStreamReader(socket.getInputStream()));
13:  PrintWriter pw = new PrintWriter(new BufferedWriter(
       new OutputStreamWriter(socket.getOutputStream())),
```

1. http://java.sun.com/j2se/
2. http://lucene.apache.org/nutch/

```
14:  while(true){
15:    String str = br.readLine();...
16:    pw.println("Message Received");
17:    pw.flush();
18:  }
19:  ...
20:  br.close();
21:  pw.close();
22:  socket.close();
23:  s.close();...
24:}
```

If we consider the above code as a piece of client code, the bold code elements are API elements, since they are imported from a third-party library, J2SE.

Definition 3. An API usage is the way to call API code, which includes its call sequences or invariants.

The above code illustrates the two types of API usages: Lines 10 to 23 show the legal call sequences to implement a server, and Line 10 shows an invariant for the `ServerSocket` method. Ernst *et al.* [17] define an invariant as a property that holds at a certain point or points in a program. When calling the above API method, programmers typically set the port number (the parameter) as `8080`. As many approaches (*e.g.*, [17]) mine invariants according to their frequencies, they can mine $parameter = \texttt{8080}$ as an invariant, when calling the above method. Except several exceptions (*e.g.*, [53], [92]), most existing approaches mine specifications from client code. In particular, API usages are extracted through the following sources:

1. Source code. Some tools (*e.g.*, [90]) statically analyze source files of client code to extract API usages. Due to the complexity of code, static approaches can produce infeasible call sequences or values. For example, an early version of MAPO [80] can extract call sequences from both `if`-clauses and `else`-clause, and thus extracts infeasible call sequences. However, it is easier for static analysis to extract all the API usages from a piece of client code.

2. Trace. Some tools (*e.g.*, [5]) instrument client code or API code. After that, they execute client code with various input values, and the instrumented code records API usages. Although API usages in traces are accurate, it can lose some API usages, due to the difficulty to prepare sufficient test cases.

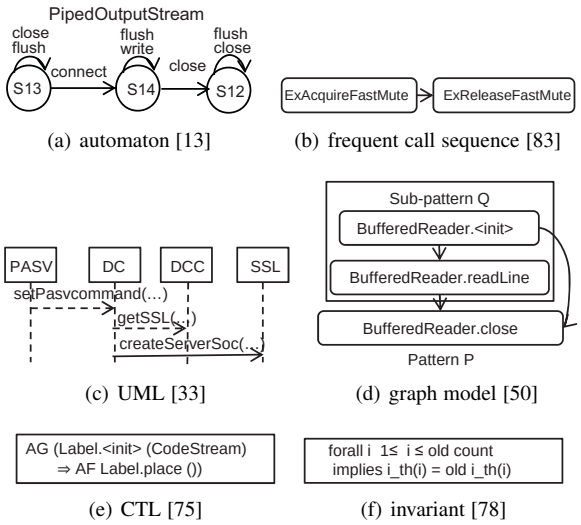
3. Bytecode. Some tools (*e.g.*, [51]) statically analyze bytecode of client code to extract API usages. It is simpler to analyze bytecode than source code, but analyzing bytecode has its unique challenges. For example, Meng and Miller [40] complain that bytecode can have non-code bytes, missing symbols, and overlapping instructions, which complicate the analysis.

All the three sources have their advantages and disadvantages. Despite of the different sources, when extracting API usages, existing tools record method sequences (*e.g.*, [5], [90]), graphs (*e.g.*, [50]), or occurrence sets [42]. Although it is easier to store call sequences, graphs can better present API usages in some cases. For example, in the above code, if we record API usages in chronological order, the called API methods in Line 12 are before the called API methods in Line 13. However, even if the order of the two lines is changed, the semantic of the code remains the same. Acharya *et al.* [2] show that graphs are natural to define the above usage, since it is a partial order.

2.2 Mining Technique

Researchers have proposed various mining techniques that mine specifications [60], and the definition is as follow:

Definition 4. A specification defines a type of legal API usages (*e.g.*, legal call sequences or invariants).



(a) automaton [13] (b) frequent call sequence [83]

(c) UML [33] (d) graph model [50]

(e) CTL [75] (f) invariant [78]

Fig. 2. Sample mined specifications

An API usage can be either legal or illegal, but common API usages are likely to be legal usages. As a result, most existing techniques are Apriori-based [4]. In this paper, we classify existing mining techniques into the following categories:

1. Sequential pattern mining. From multiple sequences of observations, the sequential pattern mining techniques [3] discover the subsequences that frequently appear in observed sequences. For mining specifications, existing approaches (*e.g.*, [90]) consider extracted call sequences as their observations, and mine frequently called sequences of API methods.

2. Grammatical inference. From multiple sequences of observations, the grammatical inference techniques [14] learn a formal grammar or a finite state machine that accounts for the characteristic of observations. For mining specifications, most approaches (*e.g.*, [5]) use the k-tail algorithm [7] to learn automata from called sequences of API methods.

3. Frequent subgraph mining. The frequent subgraph mining techniques [82] mine frequent subgraphs that appear in the observed graphs. For mining specifications, existing approaches (*e.g.*, [50]) extract graphs from client code, and discover frequent subgraphs as graph patterns. Robillard *et al.* [60] point out that graphs and automata are equivalent for mined specifications. However, their underlying techniques are different. The grammatical inference techniques recover automata from call sequences, while the graph mining techniques identify subgraphs from graphs.

4. Frequent itemset mining. From a large set of transaction items, the frequent itemset mining techniques [8] discover items whose frequencies are more than a predefined threshold. In literature, existing approaches (*e.g.*, [29], [42], [70]) mine API calls that often appear in the same piece of code or the same revision.

Besides the above major techniques, researchers have proposed other in-house techniques to mine specifications. Robillard *et al.* [60] present a comprehensive review on the research in this line. For example, Kremenek *et al.* [25] define a template for API usages, and use the factor graph techniques [35] to build specifications that fit the predefined template.

2.3 Mined Specification

Figure 2 shows six example mined specifications in the literature. In particular, Figures 2(a) to 2(e) define legal call sequences of methods, and Figure 2(f) defines legal relations of runtime values. Mined specifications have the following benefits:

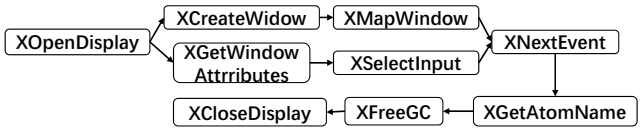


Fig. 3. A sample specification for partial and strict orders

1. Specifications concisely define API usages. As specifications summarize usages from many code samples, programmers can understand API usages without reading many code samples one by one. Although specifications can lose details, existing approaches (*e.g.*, [90]) can recommend their related code samples.

2. Specifications reflect ideal API usages. Technically, mined specifications do not exactly repeat API usages that are extracted from client code. Instead, many details can be filtered. For example, sequential pattern mining and graph mining ignore observations whose frequencies are lower than a predefined threshold. As another example, Ammons *et al.* [5] point out that their underlying k-tail algorithm [7] can ignore some sequences that do not fit the learnt automata. It is reasonable to ignore some details, since such details are relevant to only specific implementation purposes. As specifications ignore such details, they reflect ideal API usages, which are focused and easier to understand.

As shown in Figure 2, most approaches mine specifications that define legal call sequences of APIs. It needs quite different techniques to analyze call sequences and invariants. Our study focuses on call sequences to cover more relevant questions. We further discuss this issue in Section 8.

3 Research Goal

Our research goal is to provide insights on how to improve the research on APIs. Instead of specific APIs, to achieve our research goal, we need to analyze common ways to call different types of API elements. Here, API elements refer to classes, methods, and fields that are declared by API libraries and called by client code.

In this section, we break our research goal into six research questions about API usages (Section 3.1), and then analyze the significance of the research questions, as far as the research on mining specifications is concerned (Section 3.2).

3.1 Research Question

In our study, we have the following research questions:

RQ1. What is the role of API fields and static API elements?

We notice that researchers typically focus on only API instance methods. For example, all the specifications in Figure 2 define method usages, silently neglecting fields. Many researchers believe that fields are seldom used, since the information hiding principle does not recommend direct accesses on fields. However, we argue that we still need more evidences to fully understand the importance of fields and static code elements. In addition, static API methods often have quite different usages from instance API methods. For example, calling an instance method often changes the states of its declaring type, but calling a static method often does not change the states of its declaring type. The difference can lead to their different usages. A simple way to answer the problem is to count corresponding API elements, but this way is indirect and does not reflect their true usages. Instead, we analyze client code to provide our direct answers.

RQ2. Which is the best format to define API usages?

As shown in Figure 2, researchers have mined specifications in various formats (*e.g.*, frequent call sequences, automata, and graphs). However, it is still unclear which is the best format to define API usages, and in which cases. Some researchers believe that the choice of formats is subjective. However, we argue that there may be some objective measures to determine the formats. For example, Acharya *et al.* [2] proposed an approach that mines partial-order specifications, and Figure 3 shows an example of their mined specifications. The specification shows that `XFreeGC` and `XCloseDisplay` are in a strict order, and `XSelectInput` and `XMapWindow` are in a partial order. In this example, sequences naturally define strict orders, but graphs naturally define partial orders. To answer the research question, we build API graphs (Section 4.2), and count how many built graphs illustrate partial orders, *i.e.*, having branches.

RQ3. What is the right tradeoff between single-type usages and multiple-type usages?

In this paper, we define single-type API usages as usages that involve individual API classes, and multi-type API usages as usages that involve multiple API classes. Intuitively, multi-type API usages can be more useful, since corresponding specifications describe API usages of multiple classes and such usages are less documented. However, researchers often have to make a trade-off, since the multi-type analysis is more expensive than the single-type analysis. For example, when Moreno *et al.* [43] propose an approach that extracts code samples from the client code of a given method. When extracting code samples, they approach uses a static slicer [73] to compute an intra-procedural, backward slice of the given method. To answer the research question, we count how many API usages involve single types or multiple types.

RQ4. To what degree do programmers work with APIs from different libraries?

It is widely known that programmers use more than an API library in a project, but it is less unknown how programmers use APIs from different libraries. For example, it is unclear whether programmers typically use APIs within individual libraries, or use APIs from different libraries closely to implement complicated functionalities. The difference can have impacts on how to evaluate proposed approaches. Although conducting evaluations on individual libraries is clear to present results, many interesting findings may be neglected, if programmers frequently use APIs from different libraries closely in practice. To answer the research question, we count how many API usages involve single libraries or multiple libraries.

RQ5. What are the proper lengths for API usages?

It is largely unknown how lengthy API usages can be. The answer to this research question can improve existing approaches. For example, researchers (*e.g.*, [37]) proposed various code search engines that recommend code samples for APIs. If most API usages are lengthy, it can improve their effectiveness to recommend long code samples. In this paper, we present our insights by analyzing the sizes of API graphs.

RQ6. How frequently are APIs used?

Researchers still do not fully understand how frequently programmers use APIs. Thummalapenta and Xie [69] provide an indirect answer, since they find that even popular API libraries have unpopular API classes. Their results do not indicate that unpopular APIs are useless, since with the evolution of software, unpopular APIs can become popular. For example, the latest API library can implement many new APIs. When the new APIs are just released, we typically cannot find their client code, but later they can become popular. The question matters, since client code is a common source for the research on APIs. If many APIs are

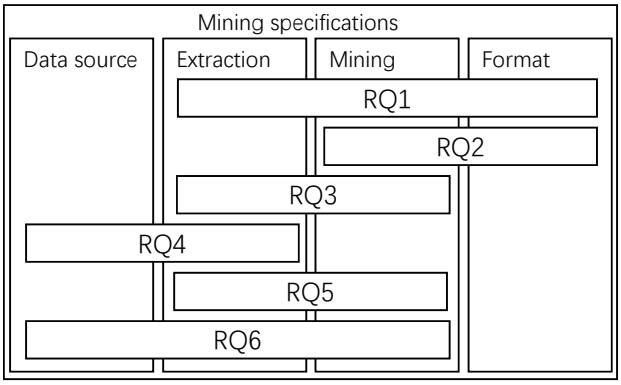


Fig. 4. The relations between RQs and mining specifications

unpopular, researchers can consider other sources. In this paper, we present the frequencies of called APIs to answer the question.

3.2 The Significance of the RQs

Figure 4 shows the significance of our research questions, in the context of mining specifications. Based on the major steps of mining specifications (see Section 2 for details), we list four key components such as data sources, extraction techniques, mining techniques, and formats of specifications. Our research questions cover all the four components:

1. Data source. RQs 4 and 6 are concerned with the data sources of mining specifications. For example, when evaluating their specification mining approaches, some researchers (*e.g.*, [18]) consider method calls of all third-party libraries as API method calls, and other researchers (*e.g.*, [2]) focus on only individual API libraries. If programmers often closely use APIs from different libraries in a code snippet, it is more reasonable to consider all third-party libraries than specific libraries in evaluations. As another example, if many APIs do not have sufficient client code for mining, it is worthy to explore other data sources (*e.g.*, [92]).

2. Extraction techniques. RQs 1, 3, 4, 5, and 6 are concerned with the extraction techniques of mining specifications. For example, when extracting API usages, most approaches ignore API fields. If fields play an important role, it is reasonable to consider API fields for extraction techniques. As another example, it needs much different techniques to extract API usages from other sources (*e.g.*, documents) than client code.

3. Mining techniques. RQs 1, 2, 3, 5, and 6 are concerned with the mining techniques of mining specifications. For example, Robillard *et al.* [60] show that only 7 out of 33 surveyed approaches mine multi-type specifications, due to the complexity of mining such specifications. If we understand in which cases multi-type usages frequently occur, we can design corresponding techniques for such cases. As another example, researchers (*e.g.*, [18], [26]) proposed approaches that combine small specifications into longer ones. The answer to the lengths of API usages can reveal the boundary of such approaches.

4. Formats of specifications. RQs 1 and 2 are concerned with the formats of mined specifications. Based on the formats of mined specifications, Zhong *et al.* [92] divide approaches for mining legal call sequences into sequence-based approaches and automaton-based approaches. In their taxonomy, mined sequences include logic and sequence diagrams. For example, Wasylkowski *et al.* [77] propose an approach that mines frequent call sequences, and then extend their approach to mine CTLs [75]. According to the taxonomy of Zhong *et al.* [92], the above two approaches are sequence-based. Robillard *et al.* [60] claim that automaton specifications and graph specifications are equivalent, so we use graph-based approaches to denote approaches that mine automata or graphs. The key difference between the two types of specifications is their capability to define partial orders. For example, in the sample code of Section 2, it is allowed to change the call sequence of Lines 12 and 13. It is quite difficult to define the call relation between the two lines as a strict order, but it is straightforward to define it as a partial order. The difference can partially explain why researchers (*e.g.*, Pradel and Gross [57]) observed that some call sequences are incidental in traces. The answers to the research questions can define proper formats under different scenarios.

Section 5 presents our major findings, and we further interpret our findings for mining specifications, in Section 6.

TABLE 1
Dataset.

Name	LOC	#AL	Description
cassandra	393,038	94	a distributed database.
derby	1,259,797	10	a relational database.
lucene	692,520	24	a search library.
jfreechart	327,104	6	a chart library.
solr	314,645	75	an enterprise search platform.
poi	567,201	15	a library for MS documents.
zoopkeeper	116,232	28	an open-source server.

4 METHODOLOGY

This section introduces our dataset (Section 4.1) and our major steps (Section 4.2).

4.1 Dataset

Table 1 shows our subjects. The jfreechart project is from SourceForge[3], and the other five projects are from Apache[4]. We selected these projects, since they are widely used. In addition, to reduce the bias on specific software, we selected projects from various categories such as databases, servers, platforms, and API libraries. Column "LOC" lists lines of code. To ensure the representativeness of our subjects, we select both small and large projects, and the largest project has more than a million lines of code. We consider source files of these subjects as client code. Column "#AL" lists used API libraries. We identify these API libraries from build configuration files of these projects. In total, these projects use 148 unique API libraries. Column "Description" lists descriptions of our subject projects.

4.2 Support Tool

We implemented a tool to support our study, and it has the following major functionalities:

1. Building dependency graphs. In the graph theory, label graphs are an important type of graphs, and its definition is as follow:

Definition 5. A labeled graph is defined as $g = \langle V, E, \mu, \nu \rangle$, where V is a set of vertices; $E \subseteq V \times V$ is a set of edges; $\mu : V \rightarrow L_V$ is a function that assigns labels to vertices; and $\nu : E \rightarrow L_E$ is a function that assigns labels to edges.

From the view of the graph theory, a dependency graph is a labeled graph, and its definition is as follow:

3. http://sourceforge.net/
4. http://www.apache.org/

Definition 6. A dependency graph is a labeled graph $g = \langle V, E, \mu_0, \nu_0 \rangle$, where V is a set of code instructions; $E \subseteq V \times V$ is a set of edges that denote data dependency or control dependency; $\mu_0 : V \rightarrow L_V$ is a function that assigns full names to vertices; and $\nu_0 : E \rightarrow L_E$ is $\emptyset$.

Our tool uses WALA[5] to build a dependency graph for each client method. As explained in the manual of WALA[6], in a dependency graph, each node denotes an instruction in a language that is close to the JVM bytecode. When building dependency graphs, WALA supports both intra-procedure and inter-procedure analysis. Its inter-procedure analysis typically produces larger dependency graphs than its intra-procedure analysis, since it explores internal structures of called client methods. Although larger graphs better reflect API usages from the view of machines, the inter-procedure analysis has negative impacts on analyzing API usages. For example, a client method (m) may be called by many client methods (M). If we apply inter-procedure analysis, the dependency graph of the client method (g) will appear in dependency graphs of all the methods in M. As a result, the frequency of g becomes high (at least $|M| + 1$ times), which can be surprising, since such usages may not be common. To reduce the bias, like other approaches (*e.g.*, [90]), our tool applies only intra-procedure analysis. The strategy does not lose any information, since it builds dependency graphs for all client methods. Section 5.7 further discusses this issue.

2. Encoding API graphs. Based on our inspection on the open questions in Section 3.1, we notice that the open questions are not concerned with the usages of specific APIs, but the general trends of using different types of APIs. To capture the usages of different APIs, our tool builds an API graph from each dependency graph, and its definition is as follows:

Definition 7. An API graph is a labeled graph $g = \langle V, E, \mu_1, \nu_1 \rangle$, where V is a set of API elements; $E \subseteq V \times V$ is a set of dependencies; and μ_1 and ν_1 are defined as follows:

$$\mu_1(v) = \begin{cases} \text{im}, & \text{if v is an instance method.} \\ \text{sm}, & \text{if v is a static method.} \\ \text{if}, & \text{if v is an instance field.} \\ \text{sf}, & \text{if v is a static field.} \end{cases}$$

$$\nu_1(\langle v_i, v_j \rangle) = \begin{cases} \text{sc}, & \text{if } v_i \text{ and } v_j \text{ are declared by the same API class.} \\ \text{sl}, & \text{if } v_i \text{ and } v_j \text{ are declared by the same library, but not the same class.} \\ \text{dl}, & \text{if } v_i \text{ and } v_j \text{ are declared by different API libraries.} \end{cases}$$

Given a dependency graph $g = \langle V, E, \mu_0, \nu_0 \rangle$, our tool builds its API graph $g' = \langle V', E', \mu_1, \nu_1 \rangle$ with the following steps:
(1) If $v \in V$ and v is an API element, it adds v to V' and assign its label as $\mu_1(v)$, and if v is not an API element, it ignores v.
(2) If $\exists \langle v_1, v_2 \rangle \ldots \langle v_n, v_m \rangle \in E$, v_1 and v_m are API elements, and $v_2 \ldots v_n$ are not API elements, it adds $\langle v_1, v_m \rangle$ to E'.
(3) It assigns labels to nodes and edges, according to Definition 7.

As the sequential rule of calling methods after calling constructors is less interesting, we ignore constructors. Indeed, as constructors are not instance methods nor static methods, our built API graphs do not include constructors.

Our tool uses ASM[7] to analyze the bytecode of API libraries. If a code element of an API library is public, it adds the code element to an API list. When analyzing client code, our tool compares the full name of a called code element with the API list to determine whether it is an API element. If a code element does not appear in the API list, we consider it as a client-code element.

3. Clustering. During the analysis process, we have to cluster data, according to our research questions. For example, to explore RQ1, we cluster client code methods into categories, based on how they call different types of API elements. In data mining, the cluster analysis [6] is the task that groups items into categories according to the similarity between items. As the cluster analysis meets our requirement, in this study, we use the traditional kmeans clustering to build clusters, and the Silhouette analysis [62] to determine the number of clusters. To reduce superficial conclusions, we remove categories whose items are fewer than a hundred after clustering. For example, when we analyze RQ1 for each API library, we do not consider API libraries whose calls are fewer than a hundred. Some data are time series data. For example, Figure 8 shows called API classes per API graph, which can be considered as time series data. To cluster such data, we use the derivative dynamic time warping [24] to calculate their similarity values.

For two time series such as $P = p_1, p_2, \ldots, p_i, \ldots, p_n$ and $Q = q_1, q_2, \ldots, q_j, \ldots, q_m$, a warping path W is a contiguous set of mappings between P and Q:
$W = w_1, w_2, \ldots, w_k, \ldots, w_u$, $max(m, n) \leq u \leq m + n$
The problem of calculating the distance between P and Q is then reduced to searching for the path that minimizes the warping cost:

$$dis(P, Q) = min \left\{ \frac{\sqrt{\sum_{i=1}^{u} w_i}}{u} \right. \quad (1)$$

Researchers have proposed various approaches that handle the problem (*e.g.*, slope weighting [64] and step patterns [45]). While most previous approaches use the Euclidean distance, the derivative dynamic time warping [24] compares the estimated derivatives of two points to calculate their distance. The estimated derivative of a point q is defined as follow:

$$d_x(q) = \frac{(q_i - q_{i-1}) + ((q_{i+1} - q_{i-1})/2)}{2} \quad (2)$$

Intuitively, an estimated derivative denotes the slope of the point to its left and right neighbors. As a result, a low distance indicates that two compared time series have similar shapes.

5 Empirical Result

We used our tool to analyze the projects in Table 1, and present our findings for the open question listed in Section 3.1.

5.1 RQ1. Field and Static Code Element

We used our tool to build an API graph for each client method in Table 1. Based on built graphs, we calculated the distribution of static methods, instance methods, static fields, and instance fields. Figure 5 shows the distribution. Based on the result, we come to the first finding:

Finding 1. Static API methods are frequently called. In total, more than 30% called elements are static API elements.

5. http://wala.sf.net
6. http://wala.sourceforge.net/wiki/index.php/UserGuide:IR
7. http://asm.ow2.org/

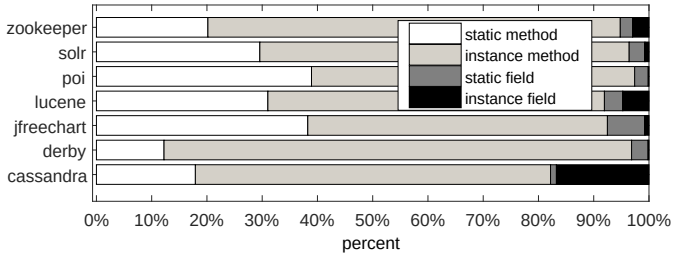
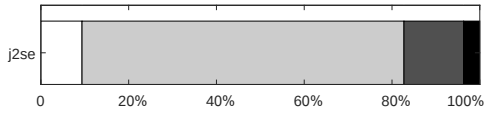


Fig. 5. The distribution of called API elements

In total, we find that about one third of called code elements are static. API libraries typically do not declare so many static code elements. For example, for J2SE, the distribution of its declared code elements is as follow:



Here, the colors of the bars are of the same meanings as Figure 5. As shown in the above bar graph, most declared code elements in J2SE are instance methods. The difference between declared code elements and called code elements indicates that static code elements are more frequently called than instance code elements. We notice that a code element is often declared as static, so that it is easily called in different contexts. The design choice may explain why static code elements are frequently called.

We further analyze the distribution of static code elements per library. For this research question, the features of a library include its occurrences of static methods, instance methods, static fields, and instance fields. Under the guide of the Silhouette analysis, we classify libraries into six categories. In each category, the occurrences of API calls (*e.g.*, the calls of static API methods) are similar. Based on the results, we find that in 80.0% of libraries, programmers call much more instance elements than static elements, and in the remaining 20% of libraries, programmers call half of their static elements and half of their instance elements roughly. An extreme library is JUnit[8], and in total, 94.6% of its called elements are static. As a testing framework, JUnit declares many static methods. For example, the `Assert` class[9] declares nine static methods, and this class is called by most test cases. This design choice can explain why so many static code elements are called for JUnit.

Based on Figure 5, we come to the second finding:

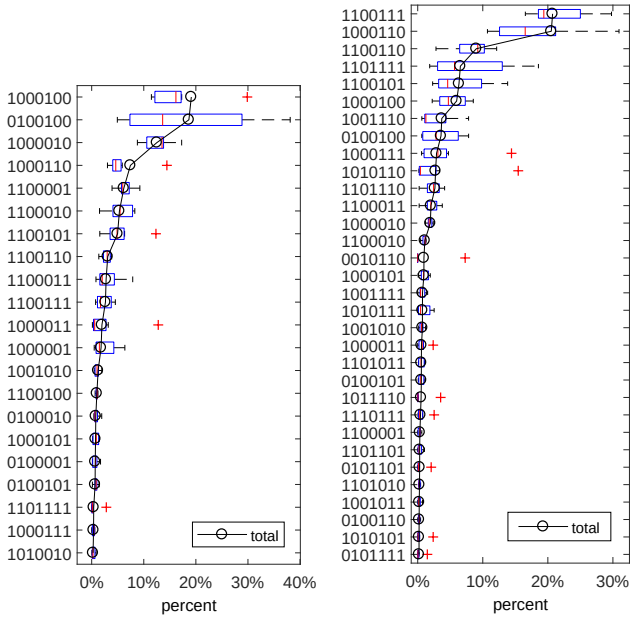
Finding 2. Methods are more frequently called than fields. In total, more than 80% called elements are methods.

We further analyzed the distribution for each library, and we find that most libraries call much more methods than fields, except the two libraries such as antlr[10] and uima[11]. For most libraries, it is reasonable to focus on only API methods, but we sill find several anomalies that need specific treatments.

To understand the role of fields and methods, we classify API graphs based on their nodes and edges. For each API graph, we generate a seven-bit key in the following format:

im	sm	if	sf	sc	sl	dl

8. http://junit.org
9. http://junit.org/javadoc/latest/org/junit/Assume.html
10. http://www.antlr.org
11. https://uima.apache.org



(a) strict order (b) partial order

Fig. 6. The classification results based on the types of nodes and edges

The bits are of the same meanings as they are in Definition 7. Each bit is either 0 or 1, where 1 denotes that the graph has the corresponding type of nodes or edges and 0 denotes no such type of nodes or edges. For example, the key, "1000100", denotes that a graph has only instance methods that are declared by the same class. During our classification, we put graphs with the same key into the same category, and use the key to name the category. Figures 6(a) and 6(b) shows the classification results for strict orders and partial orders, respectively. Here, we use *strict orders* to denote API graphs without branches, and *partial orders* to denote API graphs with branches. In the two figures, the horizontal axes list percentages from graphs in the corresponding category to the total graphs, and the vertical axes list keys of categories. The "total" line shows the result, when classify strict orders and partial orders of all the seven projects altogether. The keys of the vertical axes are in a descending order of the "total" line. The boxplots show the quartiles, when we classify strict orders and partial orders of the seven projects separatively. To save space, we do not present categories whose percents are less than 0.1%.

In Figure 6(a), the third and the fourth bits of all the top ten categories are zero. The result indicates that all strict orders in the top ten categories do not have any fields. In the contrast, in Figure 6(b), three out of the top ten categories of partial orders have fields. As a result, graph-based approaches should pay more attention to fields, whereas sequence-based approach can focus on only methods. This leads to our third finding:

Finding 3. Fields are more related to partial orders than to strict orders.

In summary, for RQ1, our results show that (1) static methods are frequently called; (2) fields are less called than methods; and (3) fields are more related to partial orders than strict orders.

5.2 RQ2. The Format to Encode API Usage

We classify API graphs into three categories, and Figure 7 shows the result. In particular, "1node" denotes graphs with only a

node; "strict order" denotes graphs without branches; and "partial order" denotes graphs with branches. In about one third of API graphs, each graph calls only an API element. From such graphs, we randomly selected 100 samples. Although our samples are insufficient to present the real distribution, after manual inspection, we identify the following cases:

1. No sequential rules. For example, the `Math` class[12] of J2SE declares tens of numeric methods. These methods do not have any sequential rules, and are often called individually by client code. Please note that we focus on legal call sequences. Even if a method does not have any sequential rules, it can follow other rules such as invariants. For example, the following code throw an exception:

```
1: public static void main(...){
2:  Random r = ...
3:  int rand = r.nextInt(-1);
4:}
```

The error message says "bound must be positive". Although the `nextInt` method does not have any sequential rules, it requires that its input value must be positive. As introduced in Section 4.2, our built API graphs do not include constructors. Although a constructor is called to construct a `Random` object, its API graph has only a node. As a result, we put it into the "1node" category.

2. Extended Types. We notice that programmers can override APIs. For example, the following code extends the `FileOutputStream` class:

```
1: public class SortedFileOutputStream extends
2:   FileOutputStream {
3:   private StringBuffer sb = null;
4:   @Override
5    public void write(byte[] b) throws IOException{
6:      if (sb == null){
7:         sb = new StringBuffer();
8:      }
9:      sb.append(new String(b, off, len));
10:  }
11:  ...
12: }
```

In the above example, the overridden method is as follow:

```
1: public void write(byte b[]) throws IOException {
2:    writeBytes(b, 0, b.length, append);
3: }
```

In the `SortedFileOutputStream` class, programmers override the above method to append the input value to a string.

3. Wrappers. We notice that programmers can implement wrappers for APIs. For example, a wrapper is as follow:

```
1: public class LogFile {
2:   private final File logFile;
3:   private FileOutputStream fileOutputStream;
3:   public void append(String log){
4:     if (fileOutputStream!=null) return;
5:     try {
6:        fileOutputStream.write(log.getBytes());
7:     } catch (FileNotFoundException e) {
8:        Log.e(...);
9:     }
10:  }
11:  ...
12:}
```

In the above code, programmers implement a wrapper, and in Lines 3 to 10, the `append` method exposes the `write` method.

When client code calls the methods of the `FileOutputStream` class, it typically follows some sequential rules (*e.g.*, `write`→ `flush` → `close`) to avoid resource leaks. From the viewpoint of an intra-procedural analysis, the usages in extended types and wrappers are partial. As a result, it is infeasible to discover full usages from extended types or wrappers, if they are not called by other client code.

12. http://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

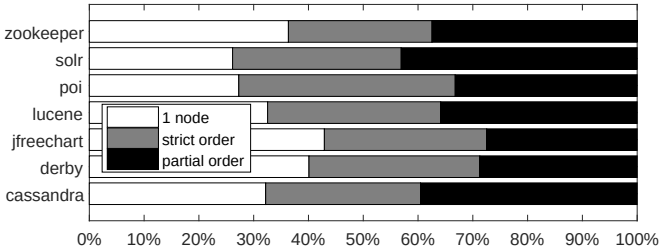
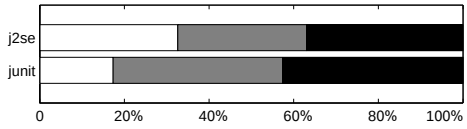


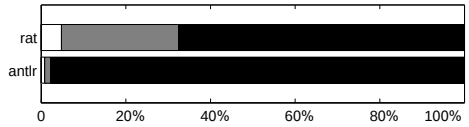
Fig. 7. The distribution of API graphs

Figure 7 shows that less than one third of API graphs illustrate strict orders. Sequence-based approaches (*e.g.*, [90]) mine frequent call sequences. Their mined specifications naturally define legal call sequences in strict orders, and it can be unnecessary to use graph mining for these API usages. Figure 7 shows that more than one third of API graphs illustrate partial orders. Graph-based approaches (*e.g.*, [50]) naturally handle these partial orders. In the contrast, sequence mining may lose valuable information of branches, and is insufficient to mine partial orders. Section 2 illustrates such an example.

We further analyze this issue for each library, and we find that most libraries follow similar patterns as shown in Figure 7. For example, the distributions of the two common libraries such as J2SE and JUnit are as follows:



Here, the colors of the bars are of the same meanings as Figure 7. However, we find that antlr and rat[13] follow a different pattern:



For the two libraries, graph mining is more suitable than sequence mining to mine their usages, since most API graphs of the two libraries are partial orders. We find that the usages of the two libraries are different. For example, antlr is a parser library, and it provides various types that provide the basic parsing functionalities. For a specific language, programmers need to extend these types. In particular, the following code parses Javascript:

```
1: class JavascriptParser extends Parser{
2:   public final ... expression()...{
3:     ...
4:     try{
5:       ...
6:       EOF2=(Token)match(input,...);
7:       retval.stop = input.LT(-1);...
8:     } catch (RecognitionException re) {
9:       reportError(re);
10:      recover(input,re);
11:    }...
12: }}
```

In the above code, `Parser`[14] is extended, and both API methods and fields are called. Typically, most API types are not designed to be extended. However, we find that many API types in antlr are purposely designed to be extended. The difference may explain why most of its usages are partial orders. Based on Figure 7, we come to our fourth finding:

Finding 4. Sequences and graphs can naturally encode the usages for a half of API graphs.

13. http://creadur.apache.org/rat
14. http://www.antlr.org/api/Java/org/antlr/v4/runtime/Parser.html

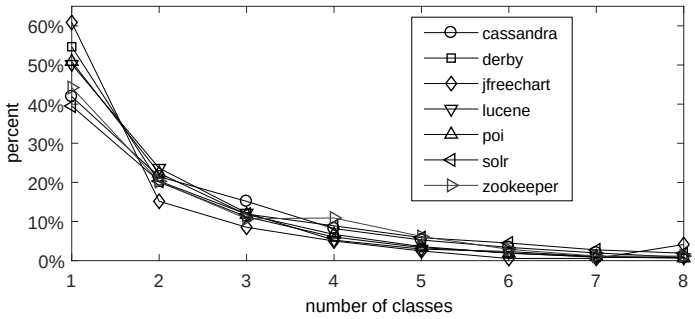


Fig. 8. The number of called API classes per API graph

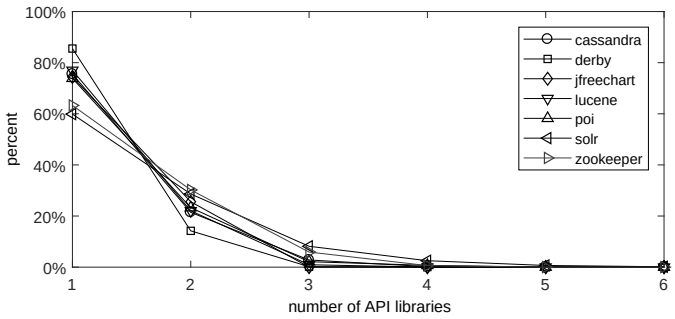


Fig. 9. The number of called API libraries per API graph

Although sequences can encode only a half of API usages, it is popular for its simplicity. As sequences have simpler structures, it is easier to mine specifications from sequences than from graphs. In addition, after specifications are mined, it is also easier to detect the violations of sequences than the violations of graphs.

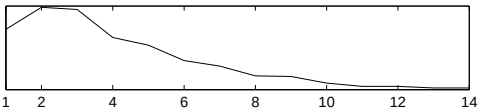
We further analyze this issue by comparing the top ten categories in Figure 6(a) with the top ten categories of Figure 6(b). We find that six categories appear in both lists. API usages in these categories include both strict orders and partial orders. For example, "0100100" ranks the second in Figure 6(a) and the eighth in Figure 6(b). It indicates that when static methods are called between classes, they can be called in either strict orders or partial orders, and both cases are common. The other four categories appear only in one list. For example, "1000010" ranks the third in Figure 6(a), but the thirteenth in Figure 6(b). It indicates that when instance methods are called among different classes, the usage is more likely a strict order than a partial order.

In summary, we find that strict orders and partial orders are half-and-half. The distribution may explain why sequence-based approaches and graph-based approaches are both popular.

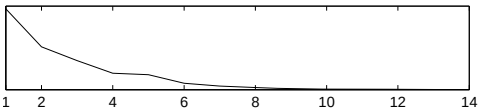
5.3 RQ3. Single and Multiple Type Usage

For each API graph, we calculated its number of involved API classes, and Figure 8 shows the result. Its horizontal axis lists number of called classes per API graph, and its vertical axis lists corresponding percent. We find that about half of the API graphs call only a class. The result indicates that multi-type usages and single-type usages are both common. In addition, we find that API usages typically do not involve many classes, since Figure 8 show that most API graphs have fewer than eight classes.

We further analyze this issue for each library. As the data in Figure 8 are time series data, we use dynamic time warping [24] to calculate the similarity values, and classify libraries into two categories: 67.8% of libraries follow the below pattern:



In the above pattern, most API graphs involves two API classes. In the remaining libraries, most API graphs involve only one API class, and the pattern is as follows:



Although most libraries follow the first pattern, the top two frequent libraries, J2SE and JUnit, follow the second pattern. As a result, the trends in Figure 8 are more like the second pattern. Based on the observations, we come to our fifth finding:

Finding 5. Single-type usages and multi-type usages are both common. For libraries such as J2SE and JUnit, single-type usages are slightly more than multi-type usages.

We further analyze this issue on the categories of Figure 6. The API graphs of single-type usages have only `sc` edges, so their keys end with "100". In Figure 6(a), the keys of three categories end with "100", and their ranks are the first, the second, and the fourteenth. In Figure 6(b), the key of only one category ends with "100", and its rank is sixth. The API graphs of multi-type usages have other key patterns. We find both Figure 6(a) and Figure 6(b) have such patterns. Our observations lead to our sixth finding:

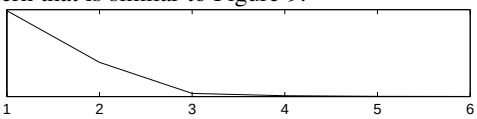
Finding 6. Single-type usages have more strict orders than multi-type usages.

In summary, according to our results, sequence-based approaches are sufficient to mine single-type specifications, but multi-type specifications need more advanced techniques. Most libraries have as many multi-type usages as single-type usages.

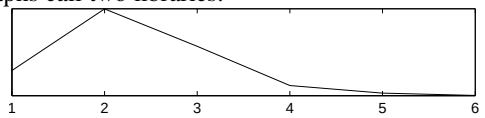
5.4 RQ4. Inter Library Usage

For each API graph, we calculated its number of called API libraries, and Figure 9 shows the distribution. Its horizontal axis lists number of called libraries for each API graph, and its vertical axis lists corresponding percent. We find that more than 80% of API graphs call only an API library in total. Section 5.2 shows that when a type extends an API type or when a type implements a wrapper for an API type, the type can implement methods that call single API method. As API graphs built from these methods do not illustrate API usages, if we ignore such cases in Figure 9, about half of API graphs call only an API library.

We further analyze this issue for each library, as we did in the previous sections. We find that 46.7% of libraries follow the below pattern that is similar to Figure 9:



The remaining libraries follow the below pattern, where most API graphs call two libraries:



Many researchers (*e.g.*, [2]) evaluate their approaches on individual libraries. Although many usages call individual libraries, about forty percent of the total usages call multiple libraries. For these usages, if we restrict our approaches on individual libraries, we can extract only partial usages, which can reduce

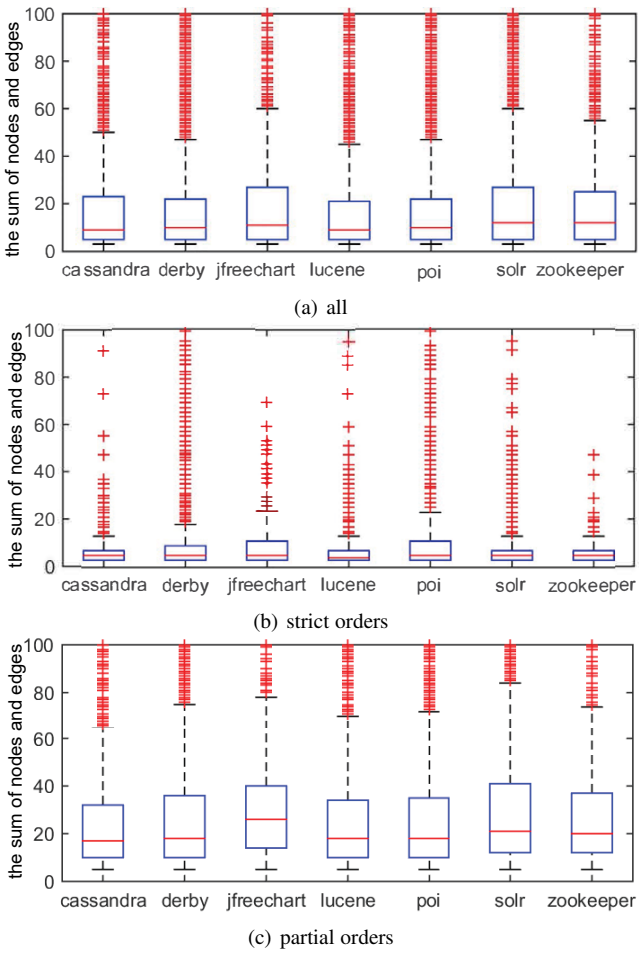


Fig. 10. The sum of nodes and edges per API graph

the effectiveness of our approach. For example, Acharya *et al.* [2] mine partial orders of API methods. When they evaluate their approach on individual libraries, they can ignore frequent call pairs between two libraries. After manual inspection, we identify the following ways when multiple libraries are called together.

1. Input values. The input value of a library can be declared by another library. The following code illustrate this way:

```
1: ByteArrayOutputStream s = ...;
2: CsvOutputArchive a = new CsvOutputArchive(s);
```

2. Thrown exceptions. A method of a library can throw exceptions that are declared by another library. In the following code, a method of Thrift[15] throws a J2SE exception.

```
1: try{
2:   TServer server = ...;
3:   server.serve();
4:   ...
5: }catch(Exception ex){
6:   ex.printStackTrace();
7: }
```

3. Extensions. A type of a library can extend types that are declared by another library. For example, `MapIterator` of Apache extends `Iterator` of J2SE, so in the following code, it allows calling the methods of J2SE:

```
1: IterableMap iterableMap = ...;
2: MapIterator it = iterableMap.mapIterator();
3: while(it.hasNext()){
4:   Object key = it.next();
5:   ...
6: }
```

15. https://thrift.apache.org

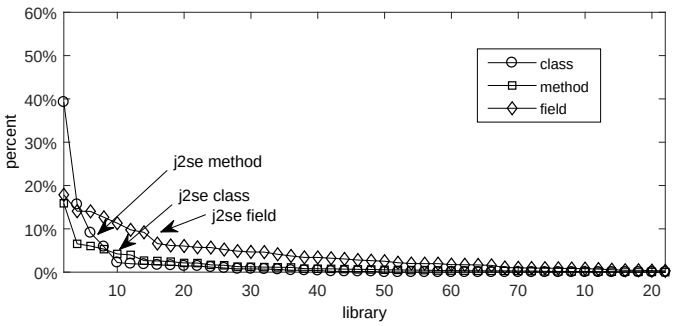


Fig. 11. The called APIs

Typically, a library is not supposed to be used with any other libraries, except J2SE. Figure 9 shows that more than 80% of usages call no more than two libraries. As shown in the above code samples, when two libraries are called, in about 90% of cases, one of them is J2SE. These observations lead to our seventh finding:

Finding 7. More than 80% of API usages call only one or two API libraries.

The result indicates that it is sufficient if researchers add J2SE to other libraries in their evaluations. Although more libraries can be called together, our result shows that such cases are rare.

5.5 RQ5. The Length of API Usage

For each API graph, we calculated its sum of edges and nodes. Here, we count both edges and nodes, since both reflect the complexity of calling APIs. Figure 10(a) shows the boxplot of results. Some methods call many API elements, so their API graphs are quite large. To better present the results, we set the upper bound of its vertical axis as a hundred. Despite of some large outliers, Figure 10(a) shows that the medians are about ten, which is small.

Gabel and Su [18] show that the number of mined real specifications become smaller, when they try to mine longer specifications. We notice that their mined specifications call about ten methods at the most. The result is consistent with ours, since the medians are also about ten. For longer specifications, it can be difficult to collect adequate API graphs.

Zhong and Su [87] analyzed thousands of bug fixes, and they find that a bug fix typically includes fewer than five API-related repair actions. Based on this finding, they suspect that an API usage is typically short. Our result provides a positive evidence to their hypothesis, since the medians are all small.

In Figure 10(b) and Figure 10(c), we build the boxplots for strict orders and partial orders separatively. Figure 10(b) shows that strict orders are shorter, and about half of strict orders are method pairs. This point explains why many approaches (*e.g.*, [83]) produce positive results, although they mine only method pairs. Figure 10(c) shows that partial orders are larger, which poses extra barriers for mining. The results lead to the eighth finding:

Finding 8. API usages are typically short, but some exceptional usages can be lengthy. In addition, partial orders typically involve more API elements than strict orders.

5.6 RQ6. The Frequency of API Usage

We use A to denote the API elements of an API library, and for each project, we use A_i to denote called API elements. Based

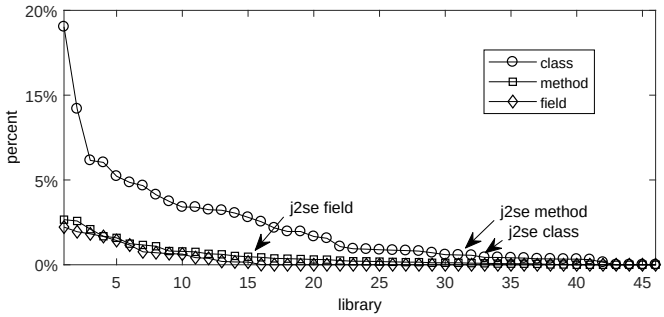


Fig. 12. The commonly called APIs

on the two definitions, we calculate the percent of called API elements as $\frac{|\bigcup A_i|}{|A|}$. Figure 11 shows the results in the descending orders of API classes, methods, and fields, respectively. We find that for most libraries, only a small portion of APIs are called. Several libraries have exceptionally high percent. After inspection, we find that they are all small libraries. For example, the noggit library[16] declares only seven classes. Although programmers often use several classes of the library, these classes already cover about half of its declared API classes.

In Figure 11, we add markers to the points of J2SE. Although our subjects have more than two million lines of code, they call only less than 10% of API elements for most libraries. A library may be called by more than one project. For these libraries, we further calculate the percent of commonly called API elements as $\frac{|\bigcap A_i|}{|A|}$. Figure 12 shows that even fewer API elements are commonly called. In Figure 12, we also use markers to denote the points of J2SE. This observation lead to our following finding:

Finding 9. Only a small portion of APIs are explicitly called by client code, and even fewer APIs are commonly called.

Our results are largely consistent with other studies (*e.g.*, [9], [38], [69]). If we analyze more projects, we anticipate that more APIs will be called, but the trend still holds. For example, Ma *et al.* [38] analyzed 39 Java projects. In total, they find that only about twenty percent of J2SE methods are called.

Although many API elements are never called, from the viewpoint of API desingers, they are still important, since they can be frequently called by other API elements. Removing such unpopular API elements can lead to the change of its functionality or even the failure of compilation.

We must warn that the real percent can be higher, since API designers may not release all public code elements as APIs. However, our results reveal a practical issue of the research on APIs. Researchers have proposed various approaches that rely on client code (*e.g.*, mining specifications [90] and recommending code samples [21]). For these approaches, it is extremely difficult to achieve high recalls of API usages, even if millions lines of code are collected. For example, Borges and Valente [9] analyzed 396 Android projects, and they complain that about forty percent of Android API methods are never called. Their approach [9] mines frequent call sequences for Android APIs. As most APIs are not frequently called, they successfully mine specifications only for fewer than ten percent of methods. Indeed, many unpopular APIs are important, and can later become popular. Furthermore, some rarely used APIs can be useful in specific tasks. For example, researchers can have to use APIs that are rarely used in production code, when they implement tools for instrumenting and debugging purposes. It can take much more effort to learn unpopular APIs, since it is rather difficult to find their existing code samples. The lack of client code places a barrier for learning API usages from client code, but also reveals the opportunities to learn their usages from other sources (*e.g.*, API documents [84], [92]).

In summary, our results show that many APIs do not have sufficient client code. To understand their usages, we have to refer to other data sources (*e.g.*, documents).

16. https://github.com/yonik/noggit

5.7 Threats to Validity

The threat to internal validity includes our technical choice of intra-procedural analysis. Although it eliminates the bias over frequently called client methods, it can split an API usage into separative graphs, which can have negative impacts on our findings. Although inter-procedural analysis can be useful in some cases, its impacts on our study are mixed. For example, Section 5.2 show that extended types and wrappers show partial usages, but neither inter-procedural analysis nor intra-procedural analysis can extract their full usages, if they are not called in other client code. As another example, inter-procedural can produce extremely long API call sequences, in which irrelevant API usages mix up. The threat could be reduced by splitting usages with API calls that indicate the beginning and the end of an API usage, but it can take nontrivial effort to identify such API calls. The threat to internal validity also includes WALA and its built system dependency graphs. WALA can fail to analyze some methods, and build wrong system dependency graphs. The thread could be reduced with more advanced tools. In addition, existing approaches can use other representations than system dependency graphs to encode API usages. As their representations and ours describe the same usages, it is feasible to interpret our findings according to the differences of representations. The threat to external validity includes that although we analyze millions of lines of code, our projects are limited and all in Java. The threat could be further reduced by introducing more subjects in future work.

6 Interpretation of our Findings

For mining specifications, we interpret our findings as follows:
1. Data source. When evaluating their approaches, if researchers select individual libraries as subjects, they should consider both their chosen libraries and J2SE (Finding 7). Although some popular APIs are intensively called, many APIs are rarely called (Finding 10). As a result, there is a strong need for mining specifications from other sources than client code.
2. Extraction techniques. Researchers should pay more attention to static API fields and methods (Finding 1). It is reasonable for existing approaches to focus on method calls (Finding 2), but when researchers mine graphs or automata, they should extract API field accesses (Finding 3). Extracting API usages for individual API libraries can lose information, but it is sufficient in many cases if the J2SE is included during analysis (Finding 7).
3. Mining techniques. Sequence-based approaches are suitable to mine single-type API usages, and graph-based approaches are suitable to mine multiple-type API usages (Findings 5 and 6). As API usages are typically short, researchers can tune their mining techniques accordingly (Finding 8). As there is a need for other data sources than client code, corresponding techniques are required to handle other types of data (Finding 10).
4. Formats of specifications. Both sequence-based approaches and graph-based approaches have their unique values in defining

multi-type and single-type API usages (Finding 4). Researchers can include fields in their specifications (Finding 2), especially when they mine graphs or automata (Finding 3).

Our study mainly focuses on mining specifications. Although this research field is intensively studied, the interpretations on this specific research field can be narrow, since the research on APIs covers much more topics than mining specifications, as we discussed in Section 1. Researchers who work on different topics can have different interpretations on our findings. As the topic is highly relevant, although we tried our best to be fair, the wide audience can have different opinions on our findings, and our interpretations are not final. Other researchers can replicate our study and provide their insights, which can deepen our knowledge on APIs and make their research more solid.

7 Related Work

Mining specifications. Ammons *et al.* [5] mine automata for APIs. Follow-up researchers [13], [18], [19], [26], [34], [53] refine their approach, and other researchers [49], [50] mine graphs. Robillard *et al.* [60] show that automata and graphs are equivalent for specifications. The research in this line can be reduced to the grammar inference problem, and can be solved by corresponding techniques (*e.g.*, the k-tail algorithm [7]). Li and Zhou [29] extract method pairs, and other researchers [63], [71] improve their approach in more complicated contexts. Engler *et al.* [16] extract frequent call sequences, and other researchers [58], [77] improve their approach with more advanced techniques. Furthermore, researchers [28], [39], [75] encode mined sequences as temporal logic. The research in this line can be reduced to sequence mining [3]. Murali *et al.* [44] lean specifications in the format of Bayesian networks. All the above approaches are concerned with call sequences. Ernst *et al.* [17] infer invariants to define rules for variables. Smith *et al.* [67] infer relations between inputs and outputs. To define more detailed API usages, researchers [36], [74] produce more informative specifications by combining frequent call sequences with invariants. Some mined specifications can be simple. Researchers [18], [26] fuse simple specifications into more lengthy and complicated ones, and other researchers [11], [56] use test cases to enrich mined specifications. Our findings are useful to improve the above approaches.

Working with APIs. Instead of proposing a mining technique, researchers also work on other perspectives of APIs. Typical scenarios include migrating API code between different languages [47], [89], searching API documents [68] or tutorials [55], evolving API client code [10], [81], detecting errors in API documents [86], classifying forum threads on APIs [93], embedding API documents with forum discussions [72], and recommending API samples [43], [94] or more effective APIs [23]. Lin *et al.* [30] learn latent locks in API code to detect deadlocks. Gu *et al.* [20] learn API usages with neural networks. Our study reveals nine findings that are useful to improve existing approaches.

Empirical Studies on APIs. Researchers conduct empirical studies to understand API usages. These studies cover the knowledge on concurrency APIs [52] or deprecated APIs [59], rules in API documents [41], the evolution of APIs [22], [66], [79], the obstacles to learn APIs [61], the link between software quality and APIs [31], the impact of API changes on forum discussions [32], the practice on specific APIs [46], the mappings of APIs [88], the adoption of trivial APIs [1], and the impact of the type system and API documents on API usability [15]. The prior studies do not fully explore our open questions. Our work focuses on a different research angle, complementing the above studies.

8 Conclusion and Future Work

It has been a hot research topic to mine specifications for APIs, and many other topics on APIs are intensively studied. However, in the research on APIs, some underlying hypotheses are still not fully explored. In this paper, we conduct an empirical study on millions lines of code to further explore API usages. Based on our results, we summarize nine findings and provide our insights to improve the future research on APIs. In our future work, we plan to explore the following two directions:

1. More API usages other than call sequences. Although most existing approaches focus on only legal call sequences, APIs can have other patterns. For example, Nguyen *et al.* [48] show that some preconditions are followed, when client code calls APIs. As another example, Zhang *et al.* [85] show that parameters of API methods follow specific patterns. Different techniques are needed to analyze these API usages. For example, it is quite difficult to determine runtime values for static analysis, so dynamic approaches are preferred. There are open questions for these usages, and an empirical study on API usages can provide insights, which needs more effort in future work.

2. The social aspects of API elements. Our built system dependency graphs have similar structures with social networks. It can be feasible to borrow existing social network analysis [65] to analyze the social aspects of API elements (*e.g.*, their roles). In future work, we plan to investigate whether such social aspects are useful to improve the research on APIs.

3. Direct improvements. Our empirical study does not make direct improvements, but our findings are useful for follow-up researchers. For example, we find that evaluating on individual libraries is insufficient and adding J2SE can cover most usages. To show its direct benefits, researchers can redo their evaluations. As another example, Legunsen *et al.* [27] complain that many false alarms are reported, when they use specifications to detect bugs. Our results show that it is imprecise to encode many multi-type usages as sequences. The imprecision can lead to false alarms. It can be worthy exploring whether it reduces false alarms, if we consider this issue. In future work, we plan to investigate how to make direct improvements with our findings.

4. Empirical studies on specific API libraries. We notice that specific API libraries can have different usages. For example, in Section 5.1, we find that JUnit has more static methods than other libraries. As a result, its usage patterns are different. As another example, in Section 5.2, we find that programmers have to extends the types of antlr, when they implement their own parsers with the library. In practice, many frameworks (*e.g.*, GEF[17]) have similar usage patterns. Instead of calling their methods, programmers have to extend their types, resulting in different usage patterns. In future work, we plan to explore usages of these specific libraries.

Acknowledgement

We appreciate the anonymous reviewers for their constructive comments. This work is sponsored by the National Key Basic Research Program of China (973 program) No. 2015CB352203, the National Nature Science Foundation of China No. 61572313, and the grant of Science and Technology Commission of Shanghai Municipality No. 15DZ1100305.

17. http://www.eclipse.org/gef/

REFERENCES

[1] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab. Why do developers use trivial packages? an empirical case study on npm. In *Proc. ESEC/FSE*, pages 385–395, 2017.

[2] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *Proc. ESEC/FSE*, pages 25–34, 2007.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. ICDE*, pages 3–14, 1995.

[4] R. Agrawal, R. Srikant, et al. Fast algorithms for mining association rules. In *Proc. VLDB*, pages 487–499, 1994.

[5] G. Ammons, R. Bodík, and J. Larus. Mining specifications. In *Proc. 29th POPL*, pages 4–16, 2002.

[6] P. Berkhin. *A survey of clustering data mining techniques*, pages 25–71. Springer Berlin Heidelberg, 2006.

[7] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, 100(6):592–597, 1972.

[8] C. Borgelt. Frequent item set mining. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(6):437–456, 2012.

[9] H. Borges and M. T. Valente. Mining usage patterns for the Android API. *PeerJ Computer Science*, 1(1):1–13, 2015.

[10] B. E. Cossette and R. J. Walker. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *Proc. 20th FSE*, pages 65–76, 2012.

[11] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *Proc. 19th ISSTA*, pages 85–96, 2010.

[12] L. de Alfaro and T. Henzinger. Interface automata. pages 109–120, 2001.

[13] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Program abstractions for behaviour validation. In *Proc. 33rd ICSE*, pages 381–390, 2011.

[14] C. De La Higuera. A bibliographical study of grammatical inference. *Pattern recognition*, 38(9):1332–1348, 2005.

[15] S. Endrikat, S. Hanenberg, R. Robbes, and A. Stefik. How do API documentation and static typing affect API usability? In *Proc. 36th ICSE*, pages 632–642, 2014.

[16] D. Engler, D. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proc. 18th SOSP*, pages 57–72, 2001.

[17] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[18] M. Gabel and Z. Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proc. 16th FSE*, pages 339–349, 2008.

[19] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *Proc. 32nd ICSE*, pages 15–24, 2010.

[20] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep API learning. In *Proc. ESEC/FSE*, pages 631–642, 2016.

[21] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proc. 27th ICSE*, pages 117–125, 2005.

[22] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente. How do developers react to API evolution? the Pharo ecosystem case. In *Proc. 31st ICSME*, pages 1–9, 2015.

[23] D. Kawrykow and M. P. Robillard. Improving API usage through automatic detection of redundant code. In *Proc. 24th ASE*, pages 111–122, 2009.

[24] E. J. Keogh and M. J. Pazzani. Derivative dynamic time warping. In *Proc. 1st SDM*, pages 5–7, 2001.

[25] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *Proc. 7th OSDI*, pages 259–272, 2006.

[26] T. D. Le, X. B. Le, D. Lo, and I. Beschastnikh. Synergizing specification miners through model fissions and fusions. In *Proc. 30th ASE*, pages 115–125, 2015.

[27] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov. How good are the specs? a study of the bug-finding effectiveness of existing Java API specifications. In *Proc. 31st ASE*, pages 602–613, 2016.

[28] C. Lemieux, D. Park, and I. Beschastnikh. General LTL specification mining. In *Proc. 30th ASE*, pages 870–875, 2015.

[29] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. ESEC/FSE*, pages 306–315, 2005.

[30] Z. Lin, H. Zhong, Y. Chen, and J. Zhao. LockPeeker: Detecting latent locks in Java APIs. In *Proc. 31rd ASE*, pages 368–378, 2016.

[31] 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 *Proc. FSE*, pages 477–487, 2013.

[32] 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 *Proc. 22nd ICPC*, pages 83–94, 2014.

[33] D. Lo and S. Maoz. Scenario-based and value-based specification mining: better together. In *Proc. 25th ASE*, pages 387–396, 2010.

[34] D. Lo, L. Mariani, and M. Pezzè. Automatic steering of behavioral model inference. In *Proc. ESEC/FSE*, pages 345–354, 2009.

[35] H.-A. Loeliger. An introduction to factor graphs. *IEEE Signal Processing Magazine*, 21(1):28–41, 2004.

[36] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proc. 30th ICSE*, pages 501–510, 2008.

[37] F. Lv, H. Zhang, J. Lou, S. Wang, D. Zhang, and J. Zhao. Codehow: Effective code search based on API understanding and extended boolean model. In *Proc. 30th ASE*, pages 260–270, 2015.

[38] H. Ma, R. Amor, and E. Tempero. Usage patterns of the Java standard API. In *Proc. 13th APSEC*, pages 342–352, 2006.

[39] S. Maoz and J. O. Ringert. GR(1) synthesis for LTL specification patterns. In *Proc. 10th ESEC/FSE*, pages 96–106, 2015.

[40] X. Meng and B. P. Miller. Binary code is not easy. In *Proc. 25th ISSTA*, pages 24–35, 2016.

[41] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini. What should developers be aware of? an empirical study on the directives of API documentation. *Empirical Software Engineering*, 17(6):703–737, 2012.

[42] M. Monperrus and M. Mezini. Detecting missing method calls as violations of the majority rule. *ACM Transactions on Software Engineering and Methodology*, 22(1):7:1–7:25, 2013.

[43] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, and A. Marcus. How can I use this method? In *Proc. 37th ICSE*, pages 880–890, 2015.

[44] V. Murali, S. Chaudhuri, and C. Jermaine. Bayesian specification learning for finding API usage errors. In *Proc. ESEC/FSE*, pages 151–162, 2017.

[45] C. Myers, L. R. Rabiner, and A. E. Rosenberg. Performance tradeoffs in dynamic time warping algorithms for isolated word recognition. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 28(6):623–635, 1980.

[46] S. Nadi, S. Krger, M. Mezini, and E. Bodden. "Jumping through hoops": Why do Java developers struggle with cryptography APIs? In *Proc. ICSE*, pages 935–946, 2016.

[47] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Statistical learning approach for mining API usage mappings for code migration. In *Proc. 29th ASE*, pages 457–468, 2014.

[48] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan. Mining preconditions of APIs in large-scale code corpus. In *Proc. 22nd FSE*, pages 166–177, 2014.

[49] H. V. Nguyen, H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen. Mining interprocedural, data-oriented usage patterns in JavaScript web applications. In *Proc. 36th ICSE*, pages 791–802, 2014.

[50] 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 *Proc. ESEC/FSE*, pages 383–392, 2009.

[51] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen. Learning API usages from bytecode: a statistical approach. In *Proc. 38th ICSE*, pages 416–427, 2016.

[52] S. Okur and D. Dig. How do developers use parallel libraries? In *Proc. 20th FSE*, pages 54–65, 2012.

[53] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language API descriptions. In *Proc. 34th ICSE*, 2012.

[54] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[55] G. Petrosyan, M. P. Robillard, and R. D. Mori. Discovering information explaining API types using text classification. In *Proc. 37th ICSE*, pages 869–879, 2015.

[56] M. Pradel and T. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *Proc. 34th ICSE*, pages 288–298, 2012.

[57] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *Proc. 24th ASE*, pages 371–382, 2009.

[58] M. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *Proc. 29th ICSE*, pages 240–250, 2007.

[59] R. Robbes, M. Lungu, and D. Röthlisberger. How do developers react to API deprecation?: the case of a smalltalk ecosystem. In *Proc. 20th FSE*, pages 76–87, 2012.

[60] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, 2013.
[61] M. P. Robillard and R. DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.
[62] P. J. Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.
[63] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui. Mining multi-level API usage patterns. In *Proc. 22nd SANER*, pages 23–32, 2015.
[64] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 26(1):43–49, 1978.
[65] J. Scott. *Social network analysis*. Sage, 2012.
[66] L. Shi, H. Zhong, T. Xie, and M. Li. An empirical study on evolution of API documentation. In *Proc. FASE*, pages 416–431, 2011.
[67] C. Smith, G. Ferns, and A. Albarghouthi. Discovering relational specifications. In *Proc. ESEC/FSE*, pages 616–626, 2017.
[68] S. Subramanian, L. Inozemtseva, and R. Holmes. Live API documentation. In *Proc. 36th ICSE*, pages 643–652, 2014.
[69] S. Thummalapenta and T. Xie. SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the web. In *Proc. 23rd ASE*, pages 327–336, 2008.
[70] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proc. 24th ASE*, pages 283–294, 2009.
[71] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *Proc. 31th ICSE*, pages 496–506, May 2009.
[72] C. Treude and M. P. Robillard. Augmenting API documentation with insights from Stack Overflow. In *Proc. 38th ICSE*, pages 392–403, 2016.
[73] N. Tsantalis and A. Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782, 2011.
[74] N. Walkinshaw, R. Taylor, and J. Derrick. Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, 21(3):811–853, 2016.
[75] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. In *Proc. 24th ASE*, pages 295–306, 2009.
[76] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. *Automated Software Engineering*, 18(3-4):263–292, 2011.
[77] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proc. ESEC/FSE*, pages 35–44, 2007.
[78] Y. Wei, C. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *Proc. 33rd ICSE*, pages 191–200, 2011.
[79] W. Wu, A. Serveaux, Y.-G. Guéhéneuc, and G. Antoniol. The impact of imperfect change rules on framework API evolution identification: an empirical study. *Empirical Software Engineering*, 20(4):1126–1158, 2014.
[80] T. Xie and J. Pei. MAPO: Mining API usages from open source repositories. In *Proc. MSR*, pages 54–57, 2006.
[81] Z. Xing and E. Stroulia. API-evolution support with Diff-CatchUp. *IEEE Transactions on Software Engineering*, 33(12):818–836, 2007.
[82] X. Yan and J. Han. CloseGraph: mining closed frequent graph patterns. In *Proc. 9th SIGKDD*, pages 286–295, 2003.
[83] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *Proc. 28th ICSE*, pages 282–291, 2006.
[84] J. Zhai, J. Huang, S. Ma, X. Zhang, L. Tan, J. Zhao, and F. Qin. Automatic model generation from documentation for Java API functions. In *Proc. 38th ICSE*, pages 380–391, 2016.
[85] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic parameter recommendation for practical API usage. In *Proc. 34th ICSE*, pages 826–836, 2012.
[86] H. Zhong and Z. Su. Detecting API documentation errors. In *Proc. SPASH/OOPSLA*, pages 803–816, 2013.
[87] H. Zhong and Z. Su. An empirical study on real bug fixes. In *Proc. 37th ICSE*, pages 913–923, 2015.
[88] H. Zhong, S. Thummalapenta, and T. Xie. Exposing behavioral differences in cross-language API mapping relations. In *Proc. ETAPS/FASE*, pages 130–145, 2013.
[89] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *Proc. 32nd ICSE*, pages 195–204, 2010.
[90] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *Proc. 23rd ECOOP*, pages 318–343, 2009.
[91] H. Zhong, L. Zhang, and H. Mei. Inferring specifications of object oriented APIs from API source code. In *Proc. 15th APSEC*, pages 221–228, 2008.
[92] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *Proc. 24th ASE*, pages 307–318, 2009.
[93] B. Zhou, X. Xia, D. Lo, C. Tian, and X. Wang. Towards more accurate content categorization of API discussions. In *Proc. 22nd ICPC*, pages 95–105, 2014.
[94] Z. Zhu, Y. Zou, B. Xie, Y. Jin, Z. Lin, and L. Zhang. Mining API usage examples from test code. In *Proc. ICSME*, pages 301–310, 2014.

Hao Zhong received his PhD degree from Peking University in 2009. His Ph.D dissertation was nominated for the distinguished Ph.D dissertation award of China Computer Federation. After graduation, he worked as an assistant professor at Institute of Software, Chinese Academy of Sciences, and was promoted as an associate professor in 2012. From 2013 to 2014, he was a visiting scholar at University of California, Davis. Since 2014, he had been an associate professor at Shanghai Jiao Tong University. He is a member of the IEEE and ACM. His research interest is the area of software engineering, with an emphasis on empirical software engineering and mining software repositories. He is a recipient of ACM SIGSOFT Distinguished Paper Award 2009, the best paper award of ASE 2009, and the best paper award of APSEC 2008.

Hong Mei received his BA and MS degrees from Nanjing University of Aeronautics and Astronautics in 1984 and 1987, respectively; and Ph.D. degree in computer science from Shanghai Jiao Tong University in 1992. From 1992 to 1994, he was a postdoctoral research fellow at Peking University. He is a professor with Shanghai Jiao Tong University, Beijing Institute of Technology, and Peking University. He was the dean of school of EECS at Peking University from 2006-2014, the vice president for research at Shanghai Jiao Tong University from 2013 to 2016, and had been the vice president for human resource and international affairs at Beijing Institute of Technology since 2016. His research interests include software engineering and system software. He is a member of Chinese Academy of Sciences, a fellow of the IEEE, CCF and TWAS.