Inferring Specifications of Object Oriented APIs from API Source Code

Hao Zhong, Lu Zhang, and Hong Mei
Key Laboratory of High Confidence Software Technologies, Ministry of Education, China
Institute of Software, School of EECS, Peking University, China
{zhonghao04, zhanglu, meih}@sei.pku.edu.cn

Abstract

API libraries are becoming increasingly popular in modern software industries because these libraries provide various methods and classes for reuse. However, as pointed out by researchers, libraries are typically difficult to use. It is desirable to infer some specifications for libraries so that programmers can learn the correct usages of these libraries. In this paper, we propose an approach to infer specifications from source code of API libraries. Our approach is based on the observation that rules in object-oriented programs can be traced from basic constraints such as memory usage, file usage, and network protocol. In addition, rules of one class spread to its dependent classes through the features of object-oriented programs such as derivation, invocation relationship, and field access among methods. Based on our approach, we implemented a prototype named Java Rule Finder (JRF) to infer specifications from source code of API libraries in Java. We conducted four case studies using JRF. The result shows that JRF infers some rules correctly. We further conducted an experiment on three open source API libraries. The results show that JRF scales well with real API libraries.

1 Introduction

In modern software industry, programmers increasingly rely on API libraries to complete their daily programming tasks. Libraries become popular because they decrease the cost and complexity of programming by providing many classes and methods that are deliberately designed for reuse. However, as pointed out by Scaffidi [15], libraries are typically difficult to learn and use.

To help programmers get familiar with the usages of libraries, many approaches have been proposed to retrieve samples or even specifications to programmers. For many widely-used libraries, existing clients can be found in open source projects. These clients can be retrieved as samples by code search engines such as Koders\(^1\). To relieve the effort to learn API usages from these samples, specification miners such as Strauss [4] can mine specifications from these retrieved clients automatically. The mined specifications describe the legal call sequences of libraries, which may be useful for programmers to understand libraries’ usages or to detect bugs in client code. However, these mining approaches need a large repository of a library’s clients to mine the library’s specifications. When a library is not widely-used or a library is a new version, its clients can be difficult to find, which limits the application of these mining approaches.

To complement the preceding approaches, we present an approach that relies on only the API libraries’ source code. Our approach is able to infer specifications when client code is unavailable or insufficient consequently. The main contributions of our work are as follows:

- **Program Rules Graph.** We propose a concise notation called program rule graph (PRG) to represent the rules that should be followed when programmers use classes provided by API libraries in the client code.

- **Iterative Rule Inference.** We propose an approach to infer specifications in the form of PRG from the source code of API libraries iteratively. The approach is based on known PRGs, collected facts, and inference strategies.

- **Evaluation.** We implemented a prototype called JRF and conducted case studies and an experiment to show its effectiveness.

The remainder of the paper is as follows. Section 2 illustrates our approach using an example. Section 3 presents the details of our approach. Section 4 presents the evaluation. Section 5 discusses issues and future work. Section 6 presents related work. Section 7 concludes this paper.

2 Example

Before presenting our approach in detail, we use an example to illustrate the basic idea of our approach. In object-
oriented programs, rules usually originate from basic constraints such as memory usage, file usage, and network protocol. In addition, classes in object-oriented programs depend on each other. Through the dependency, the rules of one class may spread to all its dependent classes.

Let us consider the code sample in Figure 1a. Class A has two array fields (i.e., a1 and a2). The field accesses of a1 and a2 appear in method1, method2, and method3. For example, in method1, a piece of memory is allocated and assigned to a1, and in method2, a value is assigned to a1. Because we should allocate a piece of memory before we set a value to it (a basic constraint of memory usage), we should invoke method1 first if we are to invoke method2, or else it can produce exceptions. Thus, our approach infers the rules of class A from the rules of an array (treated as a special class). Similarly, our approach infers the rules concerning method1, method2, and method3. As method4 invokes method2, our approach infers that the relationships among method4 and other methods of class A are similar to the relationships among method2 and other methods of class A. In addition, as class A is a concrete class, our approach infers that its usage should also follow the constraints of memory usage. In particular, our approach infers that we should allocate an instance of class A if we need to invoke its declared methods and if we have released the instance, we should not access its declared methods. Thus, our approach infers the full rules of class A.

The rules of class B depend on the rules of class A because class B has a field named b1 whose type is class A. Our approach further infers the rules of class B based on the already inferred rules of class A. In particular, from the inferred rules between method1 and method2 of class A and field accesses of b1, our approach infers that method7 should be invoked if we need to invoke method6. Our approach also infers that method5 should be invoked before we invoke method6 and method7, because our approach has inferred that the memory of class A should be allocated before the invocations of its declared methods.

Our approach may infer rules that contain conflicts. For example, from the field accesses of b2, our approach infers that method6 should not be invoked if we need to invoke method7 because method6 assigns null to b2 and method7 sets the value of b2, whereas from the field accesses of b1 our approach infers that method6 should be invoked if we need to invoke method7 because the inferred rule between method1 and method2.

After the rule of class B is inferred, our approach can further use the rule of class B to infer the rules of its dependent classes (e.g., the subclasses of class B and the classes that declare fields whose types are class B).

In the preceding example, our approach infers rules of class A from the basic memory constraints and further infers the rules of A’s dependent classes such as class B. Our approach infers rules of open source API libraries through preceding inference procedure iteratively until all the rules of involved classes are thus inferred. In particular, our approach generates initial rules based on basic constraints, and then our approach infers rules based on the initial rules and the dependencies in object-oriented programs iteratively. The details of our approach are discussed in Section 3.2.

3 Approach

3.1 Program Rule Graph

A program rule graph (PRG) is a directed graph where the vertices denote the methods and the edges denote the relationships between two methods. The vertices denote the methods of the class. One vertex has its name and its type, i.e., $v_i \in V_a, v_j \in V_a$. An edge $e = (v_i, v_j, type)$ denotes that $v_i$ should be invoked if $v_j$ should be invoked if $v_i$ is of type $type$. Thus, we use a triple $(v_i, v_j, type)$ to denote an edge, where $v_i$ and $v_j$ are the source and target vertices of the edge, and $type$ is the type edge. In this paper, we focus on two edge types ($P$ and $N$): (1) $(v_i, v_j, P)$ denotes that $v_j$ should be invoked if $v_i$ should be invoked; (2) $(v_i, v_j, N)$ denotes that $v_j$ should not be invoked if $v_i$ has been invoked.

Different types of edges may have different characteristics. We summarize three characteristics, transitive, conflictive and cycle conflictive as follows.

Transitive. For a class A and its PRG $(G(V_a, E_a))$, we call an edge type transitive when the edge type satisfies: $\forall v_i, v_j, v_k \in V_a, (v_i, v_j, type) \in E_a$ and $(v_j, v_k, type) \in E_a$ $\Rightarrow (v_i, v_k, type) \in E_a$. $P$ edges and $N$ edges are both transitive.

Conflictive. For a class A and its PRG $(G(V_a, E_a))$, we call two edge types ($type_1$ and $type_2$) conflictive when the two edge types satisfy: $\forall v_i, v_j \in V_a, (v_i, v_j, type_1) \in E_a$ and $(v_i, v_j, type_2) \in E_a$ $\Rightarrow v_i$ and $v_j$ cannot be invoked together in client code. $P$ edges and $N$ edges are conflictive.

Type cycle. A type cycle is a path such that the start vertex and the end vertex are the same and the edges in the
path are all in the same edge type.

Cycle conflictive. For a class A and its PRG \((V_a, E_a)\), we call an edge type cycle conflictive when the edge type satisfies: if \(C\) is a type cycle in \(G\) and \(\{v_1, \ldots, v_n\}\) is the set of vertices in \(C\), methods denoted by \(\{v_1, \ldots, v_n\}\) cannot be invoked together in client code. \(P\) edges and \(N\) edges are cycle conflictive.

3.2 Java Rule Finder

Figure 2 shows the overview of JRF. JRF consists of a fact collector, a PRG base, an inference engine, and a PRG viewer. In particular, the fact collector collects the support facts that are used to infer the PRGs of programs. The PRG base stores the known PRGs from basic constraints and previous inferences. The inference engine infers new PRGs based on the collected facts and the corresponding known rules from the PRG base. The newly inferred PRGs are further reconciled with the existing PRGs. The PRG viewer visualizes the PRGs in the PRG base. Some PRGs contain edge and cycle conflicts, and the PRG viewer highlights the conflicts to programmers using colors.

Figure 3 shows the detailed mechanism of our approach. It includes collected facts, basic constraints, and inference strategies. We next present the details of our inference.

Inference Engine. The inference engine infers new PRGs from the known PRGs and facts collected by the fact collector. These known PRGs are inferred from basic constraints and previous inferences. In particular, basic constraints come from memory usage, file usage, network protocol, etc. For example, we summarize the basic constraints of memory usage as follows:

- A memory should be allocated before accesses of the memory.
- A memory is not accessible after it has been released.

The basic constraints are straightforward and easy-to-understand. Programmers should follow the constraints to produce meaningful and reliable programs. From the constraints, we can infer the PRGs of basic data types such as array, list and stack. In the ‘PRGs generated from basic constraints’ module of Figure 3, we show the inferred PRG for an array (treated as a special class). In particular, we add \(\langle \text{declare, } *\text{new, } P \rangle\) because a reference of a memory should be declared before allocation of the memory. We add \(\langle *\text{new, } *\text{get, } P \rangle\) and \(\langle *\text{new, } *\text{set, } P \rangle\) because a memory should be allocated before accesses of the memory. We add \(\langle *\text{release, } *\text{get, } N \rangle\) and \(\langle *\text{release, } *\text{set, } N \rangle\) because a memory is not accessible after it has been released. Here, we treat the accesses of an array as two methods (get and set). We can further use these constraints to infer the default PRG of a concrete class. Different constraints may spread to one class and produce valuable rules.

Inference strategies are the bridges from the known PRGs to the unknown PRGs. We summarize these inference strategies from the features of object-oriented programs.

One feature of object-oriented programs is that the methods is decided by their relationships between the methods. For example, as shown in the ‘Fact’ module of Figure 3, we show the inferred PRG for the unknown PRGs. We summarize these inference strategies as follows:

1. Field strategy. Let \(A\) be a class, \(b\) a field declared in \(A\), \(B\) be the type of \(b\). The PRG of \(A\) is \(G_a(V_a, E_a)\) and the PRG of \(B\) is \(G_b(V_b, E_b)\), \(\forall v_i, v_j \in V_a, v_{is}\) denotes the first method of \(b\) invoked by \(v_i\) and \(v_{ie}\) denotes the last method of \(b\) invoked by \(v_i\). \(v_{je}\) and \(v_{je}\) are similarly defined:

   \[
   \begin{align*}
   (1) \langle v_{is}, v_{js}, \text{type} \rangle & \in E_b \Rightarrow \langle v_i, v_j, \text{type} \rangle \in E_a. \\
   (2) \langle v_{je}, v_{is}, \text{type} \rangle & \in E_b \Rightarrow \langle v_j, v_i, \text{type} \rangle \in E_a.
   \end{align*}
   \]

   The field strategy says that the relationship of two methods is decided by their first accesses and their last field accesses.

2. Derivation strategy. Let \(A\) be a class, \(B\) be a subclass of \(A\), \(S\) be the set of overridden methods of \(B\). The PRG of \(A\) is \(G_a(V_a, E_a)\) and the PRG of \(B\) is \(G_b(V_b, E_b)\):

   \[
   \begin{align*}
   (1) v \in V_a & \land v \notin S & \land v & \text{is visible to } G_b \Rightarrow v \in V_b. \\
   (2) \langle v_i, v_j, \text{type} \rangle & \in E_a & \land v_i \notin S & \land v_j \notin S
   \end{align*}
   \]
and \(v_i\) is visible to \(G_b\) and \(v_j\) is visible to \(G_b\) \(\Rightarrow\) \(\langle v_i,v_j,\text{type} \rangle \in E_b\).

The derivation strategy says that the subclass inherits the methods and the relationship between the methods from its superclass.

**Invocation strategy.** Let \(A\) be a class and the PRG of \(A\) is \(G_a(V_a,E_a)\):

1. \(\langle v_j,v_k,\text{type} \rangle \in E_a\) and \(v_j\) is the last method invoked by \(v_i\) \(\Rightarrow\) \(\langle v_i,v_k,\text{type} \rangle \in E_a\).
2. \(\langle v_k,v_j,\text{type} \rangle \in E_a\) and \(v_j\) is the first method invoked by \(v_i\) \(\Rightarrow\) \(\langle v_k,v_i,\text{type} \rangle \in E_a\).

The invocation strategy says that if a method \((v_i)\) invokes another method \((v_j)\) at first or at last, the method \((v_i)\) has its invoked method \((v_j)\)'s relationship.

The inference strategies reveal the features of object-oriented programs. We can use these features to infer new PRGs from the known PRGs and to reconcile the inferred PRGs. From different inference strategies and facts, different sub-PRGs can be inferred and they can be composed into a single PRG. For example, as shown in 'Inference' module of Figure 3, our approach infers two sub-PRGs from \(a1\) and \(a2\) for Class \(A\) shown in Figure 1. After that, our approach composes them into a single PRG and further refines the composed PRG using other facts extracted from source files and corresponding inference strategies. The refined PRG then can be verified to find the conflicts (shown in 'Verification' module of Figure 3) and reconcile with the known PRGs (shown in 'PRG database' module of Figure 3). Algorithm 1 shows the preceding mechanism in a precise way. The algorithm has two stages, the inferring stage and the reconciling stage. The inferring stage infers new PRGs from the inference strategy and collected facts whereas the reconciling stage reconcile the inferred PRGs with the existing PRGs.

Ideally, if we can build a dependency tree to describe the PRG dependency relationships among classes, we can infer the PRG of the root class first and infer the PRGs of the class’s child classes iteratively. However, such a tree is hardly to build since the dependency relationships among classes are typically quite complicated. As a result, during the inference process, the PRG of a class may become in-
consistent with the PRGs of the class’s dependent classes, and the iterative inference is in fact completed by reconciling those inconsistencies. For example, if we infer a new edge for the PRG of a class but do not update the PRGs of the class’s dependent classes, these PRGs may become inconsistent. To eliminate these inconsistencies, in Algorithm 1, we add all the PRGs with new edges to $\Delta P$ and reconcile $\Delta P$ using reconcilement algorithms. For example, Algorithm 2 shows the details to reconcile the PRG of a class and the PRG of the class’s dependent class using the field strategy. After the process of Algorithm 2, the PRG of a class and the PRGs of the classes that declare fields whose type is the class are consistent. These reconcilement algorithms form an inference cycle, and JRF thus infers new PRGs from known PRGs.

**Algorithm 1: Iterative PRG Inference Algorithm**

**Data:** $C$ is the classes in an API library; $P$ is the known PRGs  
**Result:** $P$ is the inferred PRGs  
**begin**  
$T \leftarrow \text{buildDerivationTrees}(C)$  
for $t \in T$ do  
while notAllTraversed(t) do  
$c \leftarrow \text{PostorderTraverse}(t)$  
if isConcreteClass(c) then  
$p \leftarrow \text{generateDefaultPRG}(c)$  
for $f \in c$.fields do  
$\bar{p} \leftarrow \text{inferPRGbyField}(f, P)$  
$p \leftarrow \text{compositePRG}(p, \bar{p})$  
$p \leftarrow \text{inferPRGByInvoke}(c, p)$  
$P \leftarrow P \cup \{p\}$  
$\Delta P \leftarrow \{p\}$  
while $\Delta P \neq \emptyset$ do  
$\Delta P_1 \leftarrow \text{reconcilePRGByField}(C, \Delta P)$  
$\Delta P_2 \leftarrow \text{reconcilePRGByDerivation}(T, \Delta P, P)$  
$\Delta P \leftarrow \Delta P_1 \cup \Delta P_2$  
end  
**end**

**Algorithm 2: reconcilePRGByField**

**Data:** $C$ the classes in an API library; $\Delta P$ is the input delta PRGs; $P$ is the known PRGs  
**Result:** $\Delta P'$ is the output delta PRGs  
**begin**  
$\Delta P' \leftarrow \emptyset$  
for $p \in \Delta P$ do  
for $c \in C$ do  
for $f \in c$.fields do  
if $p$.name=$f$.type then  
$p' \leftarrow \text{inferPRGByField}(f, P)$  
$P \leftarrow \text{compositePRG}(p, p')$  
$\Delta P' \leftarrow \Delta P' \cup \{p\}$  
end  
**end**

PRG Viewer. Programmers can inspect inferred PRGs through the PRG viewer. Figures 4, 5, 6, and 7 show some PRGs displayed in the PRG viewer. In particular, one vertex denotes one method. The color of a vertex varies to show the characteristic of the method: the red denotes methods with edges of conflicts; the blue denotes public methods; the green denotes protected methods; the gray denotes private methods. The edge between two vertices denotes the relationship between the two vertices (see Section 3.1 for the definitions of $P$ and $N$ edges).

PRG Base. The known PRGs are stored in the PRG base. The known PRGs include the PRGs that based on the basic constraints and the PRGs obtained from previous inferences. All the PRGs can be saved/loaded in XML.

Facts Collector. Facts are the support information to infer new PRGs and to reconcile the inferred PRGs. For example, in Figure 1, the field accesses (i.e., $a1$ and $a2$) of class $A$ are facts because they can be used to infer class $A$’s PRG. The current implementation of facts collector extracts the facts through static analysis. The collected facts include the derivation relationship among classes, the invoked methods of a method, and the field accesses of a class (*declare, *new, *destroy, and method invocations). The inference engine and the PRG base form an inference cycle in JRF. JRF carries on the inference cycle iteratively to infer PRGs. The process will not stop until all the collected facts are used to infer new PRGs.

4 Evaluation

Our evaluation intends to investigate two main research questions. One is that whether JRF can infer new PRGs from known PRGs using our mechanism (Section 4.1). The other is whether our approach scales well with real libraries (Section 4.2).

4.1 Case Studies

To prepare the case studies, we generated a default PRG for `java.util.ArrayList` and added the PRG to the rule base as the initial known PRG. The following case studies provide samples that use `java.util.ArrayList` to simulate the classes that use `java.util.ArrayList` as a field in API libraries. We checked whether JRF can infer new PRGs from the known PRG of `java.util.ArrayList` and whether JRF can detect conflicts in inferred PRGs correctly.

1: Inference Strategies. Figure 4a and Figure 4b show the source files used in this case study. Figure 4c shows the inferred PRG of class $A$ shown in Figure 4a, and Figure 4d shows the inferred PRG of class $B$ shown in Figure 4b. The label of an edge shows the edge type. JRF inferred these
PRGs based on collected facts from source code and corresponding strategies. In particular, in Figure 4c, the edges among method1, method2, method3, method4, method5, and method6 were inferred from the field strategy, whereas the edges between method7 and other methods were inferred from the invocation strategy. In Figure 4d, as class B extends class A, the PRG of class B inherits the methods and their relationships from class A except two methods: (1) the PRG of class B does not have method6 because method6 is declared as private and is not visible to class B; (2) as class B overrides method5, method5 of class B does not derive the edges from class A. This case study shows that JRF infers new PRGs from known PRGs correctly.

2: PRG Composition. From different facts and inference strategies, different sub-PRGs can be inferred. This case study investigates whether JRF can compose the sub-PRGs into one PRG correctly. Figure 5a shows the sample code used in the case study. Class A in Figure 5a declares two fields (i.e., list1 and list2). According to the field strategy, JRF inferred the sub-PRGs shown in Figure 5b from list1 and Figure 5c from list2 respectively. Figure 5d shows the composed PRG from preceding sub-PRGs by JRF. Comparing the sub-PRGs in Figure 5b and Figure 5c with the PRG in Figure 5d, we find that the PRG in Figure 5d has all the edges of sub-PRGs in Figure 5b and Figure 5c. The results show that JRF composes the sub-PRGs into one PRG correctly.

3: Conflict Detection. Although conflicts in PRGs may not be errors, extracting conflicts is valuable because methods with conflict may not be invoked together. To prepare this case study, we added one conflict to the source file and find whether JRF can detect it correctly. Figure 6a shows the source file used in the case study. Figure 6b shows the corresponding PRG inferred by JRF. JRF detected that there are two conflicting edges between method1 and method4 (colored as red). A programmer may further check the corresponding facts to get insights of the design of API libraries. The inference helps programmers to understand why they should follow a particular rule to invoke the methods.

4: Iterative Inference. This case study investigates whether JRF can infer PRGs iteratively. Figure 7a and Figure 7b show the source files used in the case study. In particular, class A declares a field named list whose type is java.util.ArrayList, and class B has a field named a whose type is A. Figure 7c shows the inferred PRG of class A by JRF, and Figure 7d shows the inferred PRG of class B. The two PRGs were both inferred through the field strategy. The case study shows that JRF iteratively infers the PRGs of class A and B from java.util.ArrayList correctly.

The four case studies confirm that JRF infers new PRGs from existing PRGs and collected facts correctly. The rule of java.util.ArrayList spreads to other classes in all the four case studies. The source files used in these case studies are simple and straightforward. However, due to the complex of object-oriented programs, the rules among the methods are usually not so straightforward. For example, the methods may hide in the superclass and the methods in the subclass may have access to the fields declared in its superclass. JRF still automatically infers their PRGs using the mechanism as described Section 3.

4.2 Experiment

Our experiment investigates whether our approach scales well with real libraries. The experiment was conducted on a PC with an Intel Pentium IV CPU 3.40GHz and 2.5G memory. Table 1 shows the results of our experiment. Column “Library” lists all the three subject libraries. Column “LOC” lists the line of code of the subject libraries. Column “Time” lists the time used for the inference in seconds. Column “PRG” lists the number of inferred PRGs. JRF generates default PRGs for all concrete classes. If an inferred
Figure 6. Conflict Detection

Figure 7. Iterative Inference

PRG contain only the edges described in a default PRG, it is not counted into the inferred PRGs because the rules described in the default PRG are already commonly known. Column “Conﬂictive PRG” lists the PRGs with conﬂictive edges and its proportion to the number of inferred PRGs. The result shows that our approach scales well in real API libraries. However, the results also show that our inferred PRGs with conﬂicts are more than it should be. We further investigate those PRGs with conﬂicts, and we ﬁnd two main limitations of JRF: the imprecision of static analysis and the descriptive capability of PRG (see Section 5 for details).

5 Discussion and Future Work

Imprecision of static analysis. We ﬁnd that some facts are diﬃcult to collect accurately. For example, a ﬁeld access of a method may hide in other methods. JRF now relies on static analysis for those facts, and some collected facts are not fully accurate due to the imprecision of our static analysis. We plan to leverage some mature static analysis techniques to collect these facts in future work. For example, Antkiewicz et al. [6] propose framework-speciﬁc models that support queries such as assignNew and assignNull with an impressive accuracy. These queries are quite useful to collect facts for our inference.

Descriptive capability of PRG. We ﬁnd that relationships between methods are typically complicated. PRG cannot describe some of these relationships accurately. For example, the following code snippet shows a class named A.

```java
public class A {
    private ArrayList list;

    public void method1() {
        list = new ArrayList();
        list2 = null;
    }

    public void method2(int index) {
        list.remove(index);
    }

    public void method3() {
        list Index;
    }
}
```

JRF infers an edge \((\text{method1, method2, N})\) for the PRG of class A from the ﬁeld strategy. The edge indicates that method2 should not be invoked if method1 is invoked. However, this edge is valid only when the parameter \(b\) is true. PRG cannot describe this relationship accurately and may thus introduce false conﬂicts. We plan to extend PRG to describe these complicated relationships between methods and develop supporting facilities for mining those extended PRGs in future work.

6 Related Work

Client-side approaches. Mining speciﬁcations from clients has long been a research focus. Engler et al. [7] use z-statistic as a support to mine frequent call sequences from client code. Acharya et al. [2] propose an approach to generate interface robustness properties from client code. Yang et al. [20] propose Perracotta to mine frequent call pairs from imperfect traces. Ammons et al. [4] propose Strauss to mine automata from program traces that are related by traditional data ﬂow dependencies. Lo and Khoo [10] further improve Strauss by introducing clustering techniques to reﬁne the traces before the mining process. Weimer and Necula [18] present an approach to mine temporal speciﬁcations to detect errors. Whaley et al. [19] propose an approach to extract models from traces and reﬁne those models using static analysis. Michail [12] proposes CodeWeb to mine association rules such as that application classes
inheriting from a particular library class often instantiate another class or one of its descendants. Li and Zhou [9] propose PR-Miner to use frequent item set mining to extract rules for detecting bugs. Mandelin et al. [11] propose Prospector to synthesize API call sequences from two types \((T_{in}, T_{out})\). Tansalarak and Claypool [17] extend the approach of Prospector by adding additional queries, ranking heuristics, and mining algorithms. Ramanathan et al. [14] use sequence mining for frequent call sequences from traces extracted statically in client code. Shoham et al. [16] mine automata from client code. Acharya et al. [1] propose an approach to mine API patterns as partial orders from client code. These approaches rely on a large client code repository, and the repository may be difficult to build when an API library is not widely used. Alur et al. [3] use a model checker to randomly generate test cases and use Angluin’s algorithm [5] to infer automata from these test cases as clients. Henzinger et al. [8] also use randomly generated test cases as clients and propose an algorithm based on witness inference to synthesize both safe and permissive interfaces from these clients. These randomly generated test cases may not reflect true usages of libraries. Our approach infers specifications from source files of libraries, and thus it is capable to infer specifications when client code is unavailable or insufficient.

**Library-side Approaches.** Nanda et al. [13] infer object typestates using inter-object references. JRF can infer new rules from existing rules and collected facts from source files of libraries, complementing these approaches.

### 7 Conclusion

To relieve the effort to learn API usages, we present JRF to infer rules from libraries’ source files. JRF is based on the observation that in object-oriented programs, rules originate from basic constraints such memory usage and file usage and rules of one class spread to other classes through the features of object-oriented program such as derivations, invocations, and field accesses among methods. We conduct four case studies on JRF. The results show that JRF infers some specifications correctly. We further conduct an experiment on three open source libraries using JRF. The results show that JRF scales well with these libraries although we still need to improve its accuracy.

### Acknowledgments

We appreciate the anonymous reviewers for their insightful comments. This research is sponsored by the National Basic Research Program of China (973) under Grant No. 2005CB321805, the High-Tech Research and Development Program of China under Grant No. 2007AA010301, and the National Science Foundation of China No. 90718016.

### References