Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

An oversampling approach for mining program specifications*#

Deng CHEN†‡[1], Yan-duo ZHANG[1], Wei WEI[1], Rong-cun WANG[2],
Xiao-lin LI[1], Wei LIU[1], Shi-xun WANG[3], Rui ZHU[1]

[1]*Hubei Provincial Key Laboratory of Intelligent Robot, Wuhan Institute of Technology, Wuhan 430205, China*
[2]*School of Computer Science and Technology, China University of Mining and Technology, Xuzhou 221116, China*
[3]*School of Computer and Information Engineering, Henan Normal University, Xinxiang 453007, China*
†E-mail: dchen@wit.edu.cn

Received Dec. 6, 2016; Revision accepted July 12, 2017; Crosschecked June 12, 2018

Abstract: Automatic protocol mining is a promising approach for inferring accurate and complete API protocols. However, just as with any data-mining technique, this approach requires sufficient training data (object usage scenarios). Existing approaches resolve the problem by analyzing more programs, which may cause significant runtime overhead. In this paper, we propose an inheritance-based oversampling approach for object usage scenarios (OUSs). Our technique is based on the inheritance relationship in object-oriented programs. Given an object-oriented program *p*, generally, the OUSs that can be collected from a run of *p* are not more than the objects used during the run. With our technique, a maximum of *n* times more OUSs can be achieved, where *n* is the average number of super-classes of all general OUSs. To investigate the effect of our technique, we implement it in our previous prototype tool, ISpecMiner, and use the tool to mine protocols from several real-world programs. Experimental results show that our technique can collect 1.95 times more OUSs than general approaches. Additionally, accurate and complete API protocols are more likely to be achieved. Furthermore, our technique can mine API protocols for classes never even used in programs, which are valuable for validating software architectures, program documentation, and understanding. Although our technique will introduce some runtime overhead, it is trivial and acceptable.

Key words: Object usage scenario; API protocol mining; Program temporal specification mining; Oversampling
https://doi.org/10.1631/FITEE.1601783 **CLC number:** TP311

1 Introduction

API protocols always exist in components or classes of object-oriented programs. They specify temporal constraints regarding the order of calls of component interface functions or public methods of classes. For example, calling peek() on java.util.Stack without a preceding push() gives an EmptyStackException, and calling next() on java.util.Iterator without checking whether there is a next element with hasNext() can result in a NoSuchElementException. Client programs that violate such protocols do not obtain the desired behaviors and may even crash the program (Alur et al., 2005). API protocols are useful for many tasks of software development, such as program documentation, understanding, validation, and testing. However, they are always implicit in

‡ Corresponding author
* Project supported by the Scientific Research Project of the Education Department of Hubei Province, China (No. Q20181508), the Youths Science Foundation of Wuhan Institute of Technology (No. k201622), the Surveying and Mapping Geographic Information Public Welfare Scientific Research Special Industry (No. 201412014), the Educational Commission of Hubei Province, China (No. Q20151504), the National Natural Science Foundation of China (Nos. 41501505, 61502355, 61502355, and 61502354), the China Postdoctoral Science Foundation (No. 2015M581887), the Key Program of Higher Education Institutions of Henan Province, China (No. 17A520040), and the Natural Science Foundation of Henan Province, China (No. 162300410177)
# A preliminary version was presented at the 27th International Conference on Software Engineering and Knowledge Engineering, Pittsburgh, USA, July 6−8, 2015
ORCID: Deng CHEN, http://orcid.org/0000-0001-6359-801X

programs and undocumented. Even when available, there is no guarantee of their consistency, completeness, and correctness.

Automatic protocol mining (APM) (Ramanathan et al., 2007; Shoham et al., 2007) is a promising approach for achieving accurate and complete API protocols. It synthesizes API protocols from interactions between components and client programs. In detail, let us assume that c is a class. To mine the API protocol of c, APM approaches first collect a large number of object usage scenarios of c from client programs through program static analysis (Wasylkowski, 2007; Wasylkowski et al., 2007) or dynamic analysis techniques (Engler et al., 2001; Dallmeier et al., 2006; Ernst et al., 2007). An object usage scenario (OUS) is a sequence of method calls, all method calls of which have the same receiver object. Obviously, an OUS of class c represents a use case about how client programs use methods of c. By taking a set of OUSs of class c as input, API protocols can be inferred based on sequential data mining algorithms.

Currently, many researchers have made significant efforts on APM and various kinds of techniques have been proposed. However, just as with any data mining technique, the sparsity of training data (OUSs) may cause inaccurate results. This phenomenon has been reflected in empirical results reported by many existing mining approaches (Li and Zhou, 2005; Chang *et al.*, 2007). Generally, the number of OUSs that can be collected from a client program is smaller than or equal to that of objects used in the program (we call this kind of OUS-collecting method a 'general OUS-collecting approach' or 'general approach' hereafter). Therefore, if a class is seldom used in a program, we will achieve a small number of OUSs. This problem can be addressed to some extent by analyzing a large codebase or more programs. For example, to mine real programming rules, Thummalapenta and Xie (2011) proposed to gather code examples from code search engines. However, this approach will significantly increase time overhead. Above all, some classes may never be used in any client programs, such as the abstract class. An abstract class exposes component interfaces and leaves its implementations to sub-classes. Since it does not subsume any implementation details, it cannot be instantiated and used directly in object-oriented programs. Consequently, we cannot collect OUSs of abstract classes from client programs, and APM approaches cannot mine their API protocols. However, abstract classes usually represent high-level abstractions and obey many common and important properties. API protocols of abstract classes are clearer, more explicit, and more compact than that of their implementing classes (Dai et al., 2014). Therefore, they are useful for program understanding. Additionally, as we will point out in Heuristic 1, sub-classes will preserve API protocols of their super-classes. Thus, API protocols of abstract classes are beneficial for program validation.

To gather abundant OUSs for APM efficiently, we propose an inheritance-based oversampling approach. Our approach is dedicated to object-oriented programs. The principle behind it is as follows: Let c' be a sub-class of c. We have the API protocol of c' that should be equivalent to or stricter than that of c. In other words, a valid OUS u' of class c' should comply with the API protocols of c' and c. Based on the above observations, our approach can derive an extra OUS u from u', which is named 'reproducing parent OUS (RP-OUS)' in this paper (for differentiation, we call the OUS u' collected by general approaches 'general OUS'). The RP-OUS u is a sub-OUS of u', which can be used to mine the API protocol of class c. Our approach is appropriate for any class including the abstract class. Theoretically, given an object-oriented program p, our technique can maximally collect n times more OUSs from p than general approaches, where n is the average number of super-classes (ANoS) of all general OUSs included in p. To investigate our technique's feasibility and effectiveness, we implemented it in our previous dynamic API protocol mining tool ISpecMiner and used the tool to conduct experiments. Experimental results show that our technique can gather 1.95 times more OUSs than general approaches. Additionally, accurate and complete API protocols are more likely to be achieved. The earlier conference version of this paper appeared in Chen et al. (2015b).

The contributions of this paper include: (1) a general oversampling approach IbO for OUSs, which is appropriate for program static analysis techniques and dynamic analysis techniques; (2) an OUS-collecting algorithm IbC, which can extract general OUSs and RP-OUSs from object-oriented programs; (3) an API protocol mining technique which can learn

the API protocol for super/abstract classes that are rarely used in programs directly; (4) investigations of the effect of our technique on mining API protocols and runtime overhead.

2 Background

2.1 API protocols

The API protocol is a kind of important program specification, which can be used for many tasks of software development. For example, according to API protocols, we can generate API documentations that can assist programmers in using APIs correctly. Additionally, we can verify a program against the API protocols. Each violation of an API protocol may be a program bug. Apart from that, API protocols can assist in generating typical test cases, which makes software testing become more efficient (Pradel and Gross, 2012). Since object-oriented programs are always designed based on the principle of high cohesion and low coupling, component interfaces or a public method of classes is likely to be data-dependent and has temporal constraints (API protocols). If all APIs included in a protocol come from the same component or class, we call the API protocol a 'single-object protocol' or 'multi-object protocol'. Fig. 1 illustrates a single-object protocol and a multi-object protocol, which are excerpted from Shoham et al. (2007) and Pradel et al. (2012), respectively. Both protocols are described using finite state automata (FSA), where states represent internal states of involved objects and transitions represent method calls. The single-object protocol (Fig. 1a) consists of five public methods, which belong to the same JDK class java.security.Signature. In contrast, methods included in the multi-object protocol (Fig. 1b) come from two different classes: java.util.Collection and java.util.Iterator.

Since object-oriented programs are always designed based on the principle of high cohesion and low coupling, single-object protocols widely exist in programs and many effective automatic mining approaches have been proposed (Dallmeier et al., 2006; 2012; Shoham et al., 2007). The task of mining precise and complete multi-object protocols is a large challenge; it requires distinguishing related objects from large sets of interacting objects accurately. On the one hand, we should consider all object interactions to mine precise and complete protocols. On the other hand, large sets of interacting objects will lead to a very high computational overhead that compromises the usefulness of API protocol mining techniques (Dai et al., 2014). For simplicity and clarity, we present our technique based on single-object protocols in this work and, unless otherwise specified, API protocols refer to the single-object protocol hereafter.

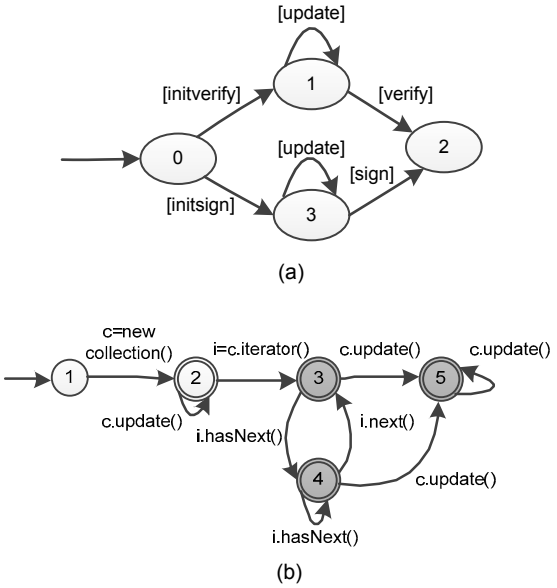

Fig. 1 Examples of a single-object protocol (a) and multi-object protocol (b)

2.2 Automatic API protocol mining techniques

Since programmers are reluctant to write temporal constraints (API protocols), many APM techniques have been proposed. Currently, a commonly used approach is synthesizing API protocols from interactions between components and client programs. This approach represents the interactions between a component and its clients by a set of OUSs. Each OUS is a sequence of method calls regarding the component. By taking a set of OUSs as input, API protocols can be inferred based on sequential data-mining algorithms. For example, Alur et al. (2005), Wasylkowski (2007), and Lorenzoli et al. (2008) mined API protocols from OUSs based on grammar inference techniques (Cook and Wolf, 1998). The grammar inference technique takes a set of sentences as input and synthesizes an FSA grammar, which can generate the

input set of sentences. Relying on such techniques, an API protocol is described using one or multiple FSAs as shown in Fig. 1. Apart from that, some researchers (Ammons et al., 2002; Chen et al., 2015b) proposed to mine API protocols from OUSs based on probabilistic models. As shown in Fig. 2, this approach works in a two-phase mode. In the first phase, API protocols described using probabilistic models, such as the Markov model, are learned from OUSs. Then, the probabilistic models are transformed to FSA by eliminating infrequent states and transitions. Compared with FSA, probabilistic models have inherent abilities to tolerate noises, which are inevitable in OUSs collected from buggy programs (Chen et al., 2015a). In the experimental section, we investigate the effect of our technique based on probabilistic models.

2.3 OUS-collecting techniques

Just as with any data-mining technique, to achieve useful API protocols, abundant training data (OUSs) should be provided. Generally, there exist two kinds of methods to collect OUSs from client programs: program static analysis (PSA) and program dynamic analysis (PDA). The PSA technique does not require running application programs. It collects OUSs from client programs by analyzing source code, bytecode, or other artifacts based on compiler technology (Chen et al., 2014; 2017). The PDA technique does not require source code as input. It gathers program execution traces (PETs) by running instrumented client programs with test cases generated manually or automatically. These two kinds of techniques have their own advantages and disadvantages and complement each other. The PSA technique (Wasylkowski, 2007; Wasylkowski et al., 2007) can cover all the program paths and can benefit from latent specifications (Engler et al., 2001), such as inferring relationships between variables based on their naming conventions, while it is tricky to deal with infeasible paths, complicated data structures, and pointer aliasing. The PDA technique (Engler et al., 2001; Dallmeier et al., 2006; Ernst et al., 2007) can be used more extensively, especially when source code is unavailable. In addition, the tricky problems in PSA can be avoided. However, its results largely depend

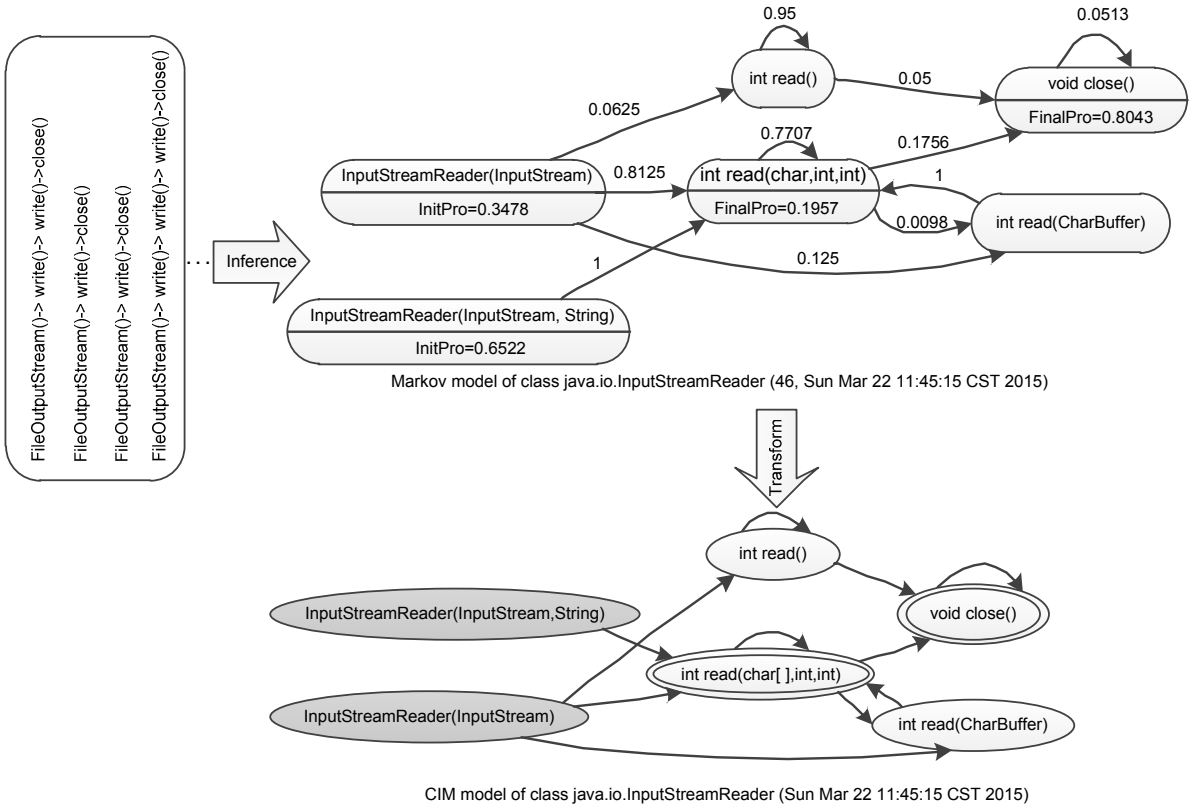

Fig. 2 The working principle of mining API protocols based on probabilistic models

upon input test cases. Whatever techniques for collecting OUSs, generally, the number of OUSs that can be gathered from a client program is smaller than or equal to that of objects used in the program. Therefore, if a class is seldom used in a program, we will achieve a small number of OUSs. Although this problem can be addressed to some extent by analyzing a large codebase or more client programs, significant time overhead will be incurred. Furthermore, it cannot work for abstract classes, which may never be used in any programs.

To resolve the above problem, we propose an inheritance-based oversampling approach for OUSs. Our technique can be used for PSA and PDA. Theoretically, it can maximally collect n times more OUSs from an object-oriented program than general approaches, where n is the ANoS of all general OUSs included in the program. Furthermore, our technique can achieve OUSs of abstract classes. Section 5 provides the details of our technique.

3 Preliminaries

To elaborate our technique clearly, we provide some preliminaries on which our work is based. In this section, we first give the formal definition of OUS. Then, we introduce the inheritance relationship of object-oriented programs and present the definition of the owner classes.

3.1 Object usage scenario

Definition 1 (Object usage scenario) Let PM(c) and $\Phi(c)$ be the sets of public methods and objects instantiated by a class c, respectively. rec(m) denotes the receiver object of a method call m. Given an object $\alpha\in\Phi(c)$, we use ous(α: c) to denote the object usage scenario (OUS) of α regarding c, which is a sequence of method calls $<m_1, m_2, \ldots, m_k, \ldots>$, $k\in\mathbb{N}$. For each method call $m_k\in$ous(α: c), we have $m_k\in$ PM(c)$\wedge$rec(m_k)=α. For notational convenience, given an OUS u, we use rec(u) and cls(u) to denote the receiver object and the class of u, respectively, i.e., rec(ous(α: c))=α, cls(ous(α: c))=c. Additionally, we use OUS(c) to denote the set of OUSs regarding class c, that is, OUS(c)={ous(α: c)|$\alpha\in\Phi(c)$}. Furthermore, we extend the notations for sets to OUSs and have the following:

$m\in u$ represents that m is a method call included in OUS u;

$\{u\}$ denotes the set of method calls included in OUS u, i.e., $\{u\}=\{m|m\in u\}$;

$<m_1, m_2>\in u$ represents that $<m_1, m_2>$ is a pair of method calls included in OUS u;

$|u|$ denotes the length of OUS u, that is, the number of method calls included in u.

In words, an OUS records the interactions (specifically, the order of method calls) between a class and its client programs. Given a set of OUSs regarding a class, the API protocol of the class can be learned based on sequential-data mining techniques. To further clarify the concept of OUSs, we present an example that illustrates how to extract OUSs from a contrived program.

Example 1 (Extracting OUSs from programs) Consider the contrived Java program shown in Fig. 3, which reads and writes data through JDK classes FileInputStream and FileOutputStream. Assume that the loop will iterate twice. According to Definition 1, we can gather the following OUSs from the program:

(1) ous(fis: FileInputStream): <FileInputStream(), read(), read(), close()>;

(2) ous(fos: FileOutputStream): <FileOutputStream(), write(), write(), close()>.

```
1 FileInputStream fis=new FileInputStream("filepath");
2 FileOutputStream fos=new FileOutput-
    Stream("filepath");
3 byte[] buffer=new byte[1024];
4 int count=0;
5 while((count=fis.read(buffer))!=-1) {
6   fos.write(buffer, 0, count);
7 }
8 fis.close();
9 fos.close();
```

Fig. 3 An example of Java program

As we can see, both OUSs consist of four method calls. The method calls included in OUS 1) are called upon the object fis (receiver object), which is instantiated by class FileInputStream. Similarly, the receiver object and class of OUS 2) are fos and FileOutputStream, respectively.

From Definition 1, we have

$$\left|\mathrm{OUS}(c, p)\right| \le \left|\Phi(c, p)\right|, \tag{1}$$

where OUS(c, p) denotes the set of OUSs collected from a program p regarding class c. $\Phi(c, p)$ represents the set of objects instantiated by class c in program p. Additionally, if a class is seldom used in a program, insufficient OUSs regarding the class will be achieved. What is worse is that, if a class has never been instantiated in a program (e.g., abstract classes), we can achieve nothing regarding the class.

3.2 Inheritance relationship of object-oriented programs

Let e be a class. We use PM(e) to denote the set of public methods of e, which can be accessed outside of e. If c is a sub-class of e, we have PM(c)$\supseteq$PM(e); that is, a sub-class will inherit all public methods from its super-classes (in this work, we consider only public methods because API protocols always subsume public methods). Additionally, we have the following definition regarding the inheritance relationship:

Definition 2 (Owner class) Let m be a public method of class c, i.e., $m\in$PM(c). We call the super-class e of c that satisfies requirement $m\in$PM(e) an owner class of m. All owner classes of m from c form a set, which is defined as follows:

$$\mathrm{ONRCLS}(m,c)=\forall_{e\in\mathrm{SUPCLS}(c)}\left(m\in\mathrm{PM}(e)\right), \quad (2)$$

where SUPCLS() denotes the set of super-classes.

Note that not all super-classes are owner classes. Given a method $m\in$PM(c), we have ONRCLS(m, c)$\subseteq$SUPCLS(c). Consider the inheritance tree regarding classes A_0, A_1, B_0, B_1, B_2, and X shown in Fig. 4. Assume that $m_x\in$PM(X) is a method inherited from class B_1. We should have $m_x\in$PM(B_2), $m_x\in$PM(B_1). According to Definition 2, we have ONRCLS(m_x, X)=$\{B_2, B_1\}$, while SUPCLS(X)=$\{A_1, A_0, B_2, B_1, B_0\}$.

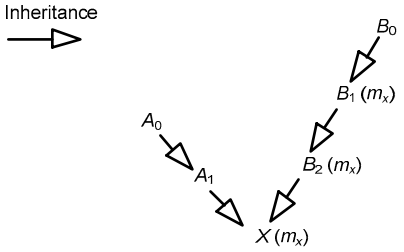

Fig. 4 An example of the inheritance tree

4 Motivating example

From Section 3.1, we can see that insufficient OUSs will be achieved if a class is seldom used in a program. In this section, we present an example that motivates our technique of oversampling OUSs.

Consider the top program shown in Fig. 5b, which reads data from a file via JDK class java.io.FileReader. If the loop iterates twice, an OUS ous(rd: FileReader) as shown in Fig. 5c will be extracted from the program. On the other hand, via substituting class FileReader with its super-class java.io.InputStreamReader (the inheritance relationship is shown in Fig. 5a), we can derive a new program as shown in the bottom of Fig. 5b. The derived program is valid and

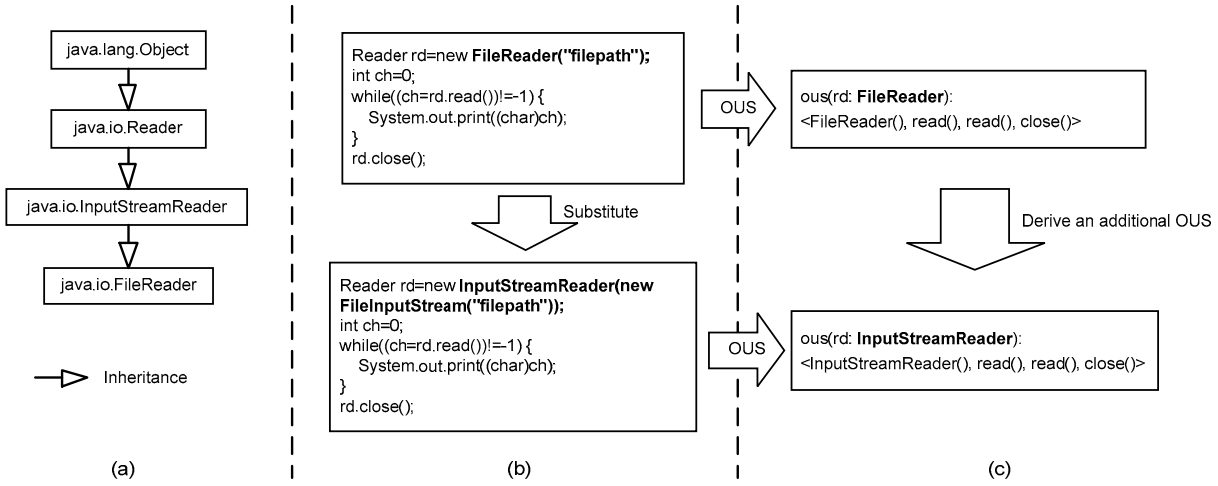

Fig. 5 Motivating example: (a) inheritance relationship; (b) class substitution; (c) OUS derivation

functionally equivalent to the original program. Additionally, from the derived program, we can extract an OUS ous(rd: InputStreamReader) as shown in Fig. 5c. The above phenomenon gives us an inspiration that we may derive an OUS regarding a class from OUSs of its sub-classes.

Note that substituting a super-class for sub-classes may not achieve a valid program because a super-class may not be an owner class of all invoked methods. For example, if we substitute the class FileReader in the top program of Fig. 5b with the super-class java.lang.Object, we will achieve an invalid program because the super-class is not an owner class of the invoked methods read() and close(). In this case, we should extract common methods from an OUS and form a derived OUS. Section 5 provides details of our technique.

5 Our technique

5.1 Inheritance-based oversampling of OUSs

To achieve sufficient OUSs for APM, we propose an oversampling approach based on the inheritance relationship among classes. Given an OUS u, our inheritance-based oversampling (IbO) approach can derive multiple OUSs from u.

Our technique is based on the following heuristic in terms of the inheritance relationship among classes:

Heuristic 1 Let M and c be a set of methods and a class, respectively. We use $\rho(M, c)$ to denote the API protocol regarding M imposed by c. Given a sub-class c' of c, we have $\rho(\mathrm{PM}(c), c') \preceq \rho(\mathrm{PM}(c), c)$, where $\preceq$ means that protocol $\rho(\mathrm{PM}(c), c')$ is equivalent to or stricter than $\rho(\mathrm{PM}(c), c)$.

In other words, the implementation of a sub-class should not violate the API protocols imposed by its super-classes.

We make the heuristic based on the following literature: (1) From the perspective of data abstraction and hierarchy (Liskov, 1987), a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra. (2) According to the substitution property (Bruce and Wegner, 1986), given two types S and T, if for each object $\alpha \in \Phi(S)$ there is an object $\beta \in \Phi(T)$ such that for all programs P defined in terms of T, the behavior of P is unchanged when α is substituted for β, then S is a subtype of T.

The above data abstraction and hierarchy principles are supported by linguistic mechanisms in many object-oriented programming languages, such as Simula 67, CLU, Smalltalk, and Java. A typical case in Java language is the exception handling mechanism. In Java programs, a method can incur exceptions through the throw statement. What is interesting is that, when a method is re-implemented in a sub-class, the exceptions thrown in super-classes should be inherited. For example, assume that E is the set of exceptions thrown by a method m defined in class A. When we overwrite the method m in a sub-class A' of A, all exceptions in E should be thrown; that is, A' should not violate restrictions on exceptions imposed by A. Our case is similar to the exception handling mechanism. What is different is that we consider the constraints regarding the order of calls of methods rather than the exceptions.

In conclusion, according to the data abstraction and hierarchy principle, we know: a subtype should not violate the API protocols imposed by its supertypes regarding inherited methods; otherwise, the substitution property cannot be satisfied. What may occur is that the subtype has a stricter API protocol than supertypes.

Relying on the above analysis, we reach the following theorem:

Theorem 1 (Existence of reproducing parent OUSs) Assume that c' is a sub-class of c. PM(c) denotes the set of public methods of class c. For simplicity, we use $\rho(c)$ to denote $\rho(\mathrm{PM}(c), c)$. Given an OUS $u' \in \mathrm{OUS}(c')$ that satisfies $\rho(c')$, there exists a reproducing parent OUS (RP-OUS)

$$u :\prec m_1, m_2, m_i, \ldots, m_j, \ \ldots >, \quad i, j \in \mathbb{N}, \tag{3}$$

$m_i, m_j \in \mathrm{PM}(c) \wedge <m_i, m_j> \in u'$, such that u satisfies $\rho(c)$.

Proof (Proof by contradiction) Given a pair of method calls h and an API protocol ρ, we use h NPT ρ to denote that h cannot be accepted by ρ. Assume that u does not satisfy $\rho(c)$. We should have

$$\exists < m, m' > \in u, \ < m, m' > \ \mathrm{NPT} \rho(c). \tag{4}$$

On the other hand, from Theorem 1, we have

$$\forall < m_i, m_j > \in u, \quad < m_i, m_j > \in u'. \tag{5}$$

Thus, we have

$$\exists < m, m' > \in u',\ < m, m' > \ \mathrm{NPT}\rho(c). \quad (6)$$

Since m, $m' \in \{u\}$, we have

$$\exists < m, m' > \in u',\ < m, m' > \ \mathrm{NPT}\rho(\{u\}, c). \quad (7)$$

On the other hand, according to Heuristic 1, we have $\rho(\{u\}, c') \preceq \rho(\{u\}, c)$. Therefore, we have

$$\exists < m, m' > \in u',\ < m, m' > \ \mathrm{NPT}\rho(\{u\}, c'). \quad (8)$$

Since $\{u\} \subseteq \mathrm{PM}(c')$, we have

$$\exists < m, m' > \in u',\ < m, m' > \mathrm{NPT}\rho(c'). \quad (9)$$

Consequently, we find that u' does not satisfy $\rho(c')$, which is a contradiction.

Theorem 1 implies an approach to derive an extra RP-OUS u from u', where u can be used to train the API protocol of a super-class. For notational convenience, we denote u by rp-ous(u', c), where u' is called the source OUS of u, and c is the class of u. Note that an RP-OUS and its source OUS have the same receiver object and different classes, that is, rec(u)=rec(u'), cls(u)≠cls(u'). All RP-OUSs derived from u' form a set

$$\mathrm{RP\text{-}OUS}(u') = \{u \mid \text{rp-ous}(u', c) \wedge c \in \mathrm{SUPCLS}(\mathrm{cls}(u'))\}. \quad (10)$$

To differentiate between RP-OUSs, we call the OUS that can be extracted from a program directly (i.e., the OUS without source OUSs) 'general OUS'. Additionally, we come to the following conclusion with respect to RP-OUSs and general OUSs:

Theorem 2 (Maximum number of RP-OUSs) Let G be the set of general OUSs gathered from a program P. $R = \bigcup_{u \in G} \mathrm{RP\text{-}OUS}(u)$ is the set of RP-OUSs achieved from P. We have $|R| \le d \times |G|$, where

$$d = \sum_{u \in G} \left|\mathrm{SUPCLS}\left(\mathrm{cls}(u)\right)\right| \Big/ |G|$$

is the ANoS of all OUSs included in G (a super-class of OUS u refers to that of cls(u)).

Proof According to Eq. (1), we have

$$\forall u \in G,\ \left|\mathrm{RP\text{-}OUS}(u)\right| \le \left|\mathrm{SUPCLS}\left(\mathrm{cls}(u)\right)\right|. \quad (11)$$

By summing up the total number of RP-OUSs and super-classes of u through G, we have

$$\sum_{u \in G} \left|\mathrm{RP\text{-}OUS}(u)\right| \le \sum_{u \in G} \left|\mathrm{SUPCLS}\left(\mathrm{cls}(u)\right)\right|. \quad (12)$$

Obviously, according to the characteristics of the set union operation, we have

$$\left|\bigcup_{u \in G} \mathrm{RP\text{-}OUS}(u)\right| \le \sum_{u \in G} \left|\mathrm{RP\text{-}OUS}(u)\right|. \quad (13)$$

Based on the above analysis, we have

$$\left|\bigcup_{u \in G} \mathrm{RP\text{-}OUS}(u)\right| \le \sum_{u \in G} \left|\mathrm{SUPCLS}\left(\mathrm{cls}(u)\right)\right|, \quad (14)$$

that is,

$$|R| \le d \times |G|. \quad (15)$$

In words, the RP-OUSs that can be achieved from a program is maximally n times the number of general OUSs, where n is the ANoS of all general OUSs included in the program. The maximum value is achieved when each super-class corresponds to an RP-OUS. However, as illustrated in Example 2, this case may not be satisfied in practice.

Example 2 (Deriving RP-OUSs from general OUSs) Consider the Java programs illustrated in Fig. 6. The four classes A0, A1, A2, and A3 have the following inheritance relationships: class A1 inherits from A0, class A2 inherits from A1, and class A3 inherits from A2. Since a sub-class will inherit public methods of its super-classes, the public methods subsumed by each class are listed in Table 1 (methods inherited from super-classes are in italic). Assume that we have an OUS $u' \in$ OUS(A3): <m11, m12, m31, m32, m20, m23, m33>. According to Theorem 1, we can achieve the following RP-OUSs:

rp-ous(u', A1): <m11, m12>;

rp-ous(u', A2): <m11, m12, m20, m23>.

As we can see, two RP-OUSs are derived from u', corresponding to super-classes A1 and A2. As to super-class A0, since OUS u' does not subsume any methods of A0, we cannot derive an RP-OUS from u' regarding this class.

In conclusion, the number of OUSs that can be extracted from a program by general approaches is smaller than or equal to the number of objects used in the program. Relying on the inheritance relationship

among classes, our oversampling technique can maximally derive n times more OUSs (RP-OUSs), where n is the ANoS of all general OUSs included in the program.

Table 1 Public methods of classes with inheritance relationships

Class	Public methods
A0	m01, m02, m03
A1	*m01*, *m02*, *m03*, m11, m12, m13
A2	*m01*, *m02*, *m03*, *m11*, *m12*, *m13*, m21, m22, m23
A3	*m01*, *m02*, *m03*, *m11*, *m12*, *m13*, *m21*, *m22*, *m23*, m31, m32, m33

```
public class A0{
    public A0(){}
    public void m01(){}
    public void m02(){}
    public void m03(){}
}
```
(a)

```
public class A1 extends A0{
    public A1(){}
    public void m11(){}
    public void m12(){}
    public void m13(){}
}
```
(b)

```
public class A2 extends A1{
    public A2(){}
    public void m21(){}
    public void m22(){}
    public void m23(){}
}
```
(c)

```
public class A3 extends A2{
    public A3(){}
    public void m31(){}
    public void m32(){}
    public void m33(){}
}
```
(d)

Fig. 6 Program examples illustrating the inheritance relationship among classes: class A1 (b) extends class A0 (a); class A2 (c) extends class A1 (b); class A3 (d) extends class A2 (c)

5.2 Inheritance-based OUS collection

In this subsection, we integrate our IbO technique with PDA and propose an inheritance-based OUS collection (IbC) algorithm. IbC supports IbO and works in an online mode; that is, it proceeds with each method event of PETs sequentially. Since it does not require loading all PETs into memory at one time, minimum space overhead will be incurred.

5.2.1 Program execution traces

To simplify the introduction of our technique, we neglect technical details about PDA. Our IbC algorithm directly takes PETs, the output of PDA techniques, as the input. A PET consists of a sequence of method events. Each method event encapsulates all necessary information regarding a method call. In this study, we describe it based on the following format: <MT>:<TS>:<TID>:<CN>:<MN>:<OID>.

<MT> is the type of method event, including the general method event (type 'G') and the reproducing parent method event (RP method event, type 'R'). The general and RP method events are the basic elements of general OUSs and RP-OUSs, respectively. Generally, method events included in a primitive PET are general method events.

<TS> is the time stamp of method events, the value of the system timer at the time when a method event occurs.

<TID> is the identifier of the thread that raises a method event.

<CN> is the full qualified name of the class defining the entered method.

<MN> is the signature of the entered method.

<OID> is the identifier of the receiver object referenced to by this. It is zero if the entered method is static. To simplify the introduction of our technique, we exclude the static method from consideration.

For notational convenience, given a method event e, we use $e.t$ ($t \in$ {<MT>, <TS>, <TID>, <CN>, <MN>, <OID>}) to denote the above attributes regarding e.

5.2.2 Extracting OUSs from PETs

Given a PET t, the objective of IbC is to categorize the set of method events included in t into different groups. Each ordered group corresponds to an OUS, which consists of method events with the same attributes <TID>, <CN>, and <OID>. IbC addresses this classification problem based on the following function:

$$f(e) = \text{bkdrhash}(e.\text{TID} + e.\text{CN} + e.\text{OID}), \quad (16)$$

where e is a method event, bkdrhash() is a commonly used hash function proposed by Brian Kernighanm and Dennis Ritchie (Kernighan and Ritchie, 1988; Skala and Petruska, 2014). The hash function can uniformly map a string to an integer. Since method events included in an OUS have the same <TID>, <CN>, and <OID>, we use the results of $f()$ to uniquely label a group (OUS). After the classification, OUSs can be achieved by ordering method events in each group in terms of the <TS> attribute (this step is not necessary if we classify method events sequentially). The above method can extend to RP-OUSs straightforwardly. In detail, for each method event e, we derive a set of RP

method events regarding e as follows:

$$\begin{aligned}\mathrm{RPME}(e)=\{e' \mid\ & e'.\mathrm{MT}='R' \wedge e'.\mathrm{TS}=e.\mathrm{TS}\\ & \wedge e'.\mathrm{TID}=e.\mathrm{TID} \wedge e'.\mathrm{MN}=e.\mathrm{MN}\\ & \wedge e'.\mathrm{OID}=e.\mathrm{OID} \wedge e'.\mathrm{CN}=p,\\ & p \in \mathrm{ONRCLS}(e.\mathrm{MN}, e.\mathrm{CN})\}.\end{aligned} \tag{17}$$

For notational convenience, given an RP method event $e' \in \mathrm{RPME}(e)$, we call e the source method event of e'. The RP method event is similar to the general method event except for <MT> and <CN> attributes (the <MT> value of an RP method event is 'R' rather than 'G', and the <CN> value of an RP method event is an owner class of its source method event's <MN> value). By applying the same classification strategy used for general method events to RP method events, we can achieve RP-OUSs.

The outline of our IbC algorithm is presented in Algorithm 1. As we can see, it leverages a data structure H to store OUSs. The data structure is a set of queues, which are organized into a hash table (Fig. 7). Each entry is a key-value pair. The key uniquely identifies an OUS, which is computed based on Eq. (2). The value is a queue used to save method events of the OUS. Relying on the hash table, IbC processes method events in a PET sequentially. For each method event, it computes the identifier cid of the OUS to which the method event belongs and then pushes the method event into the corresponding queue H[cid]. Additionally, to achieve RP-OUSs, for each method event e, IbC derives the set of RP method events regarding e by calling function RPEventSet(e). After that, the set of RP method events are added to the end of the input PET, and they will also be grouped into the hash table.

Algorithm 1 Inheritance-based OUS extraction

Input: t, a PET achieved by the PDA technique.
Output: S, the set of OUSs implicit in t.
Notation: cid, the identifier of a group (OUS); H, a hash table, where the key is the identifier of a group, and the value is a queue used to save method events; E_{rp}, the set of RP method events derived from a general method event.
Methods:

```
1   for each method event e∈t do
2     cid←bkdrhash(e.TID+e.CN+e.OID)
3     Push e into the queue H[cid]
4     if e.MT="G" then
5       E_rp←call RPEventSet(e)
6       Add E_rp to the end of t
7     end if
8   end for
9   Copy data from H into S
10  Output S
```

Given a general method event e, function RPEventSet() derives the set of RP method events from e based on the following approach. First, it computes the set of owner classes of e.MN from e.CN by calling function OwnerClassSet(e). Then, for each owner class, an RP method event is composed and output. From the outline of function OwnerClassSet(), we can see that a queue Q is used. With the help of the queue, function OwnerClassSet() finds owner classes along the inheritance tree from the bottom up using a breadth-first searching approach. In detail, it first pushes all direct super-classes of e.CN into Q. Then, for each class c popped from Q, it checks whether c is an owner class. If c is an owner class, we add it to the output set W and push all its direct super-classes into Q. Note that if c is not an owner class, it is not necessary to continue checking its super-classes, because it is certain that they are not owner classes either.

Function RPEventSet(e)

Parameter: e, a general method event.
Output: E_{rp}, the set of RP method events derived from e.
Notation: W, the set of owner classes of e.
Methods:

```
1. W←call OwnerClassSet(e).
2. for each owner class c∈W do
3.   Compose an RP method event e', such that e'.MT
     ="R"∧e'.TS=e.TS∧e'.TID=e.TID∧e'.MN
     =e.MN∧e'.OID=e.OID∧e'.CN=c
4.   Add e' into E_rp
5. end for
6. Output E_rp
```

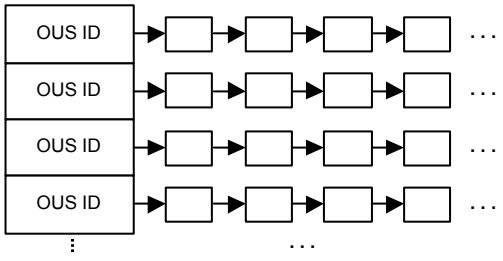

Fig. 7 The data structure used by IbC

Function OwnerClassSet(e)

Parameter: e, a general method event.
Output: W, the set of owner classes of e.MN from e.CN, i.e., ONRCLS(e.MN, e.CN).
Notation: Q, a queue used to save classes.
Methods:

```
1.  Find all direct super-classes of e.CN and push them into Q
2.  while Q is not empty do
3.    Pop the top class c from Q
4.    if e.MN∈PM(c) then
5.      Add c into W
6.      Find all direct super-classes of c and push them into Q
7.    end if
8.  end while
9.  Output W
```

5.2.3 Time and space complexity

As illustrated in Algorithm 1, the IbC algorithm consists mainly of a loop, which processes method events in a PET sequentially. Let t be the input PET. The loop will iterate $|t|$ times. For each general method event $e \in t$, IbC calls function RPEventSet() to compute the set of RP method events. The RPEventSet() function comprises a function call of OwnerClassSet() and a loop which iterates through all owner classes. In the worst case, the number of owner classes equals that of super-classes. Thus, let d be the ANoS of all method events (a super-class of a method event e refers to that of e.CN). The time complexities of function RPEventSet() and IbC should be $O(2\times d)$ and $O(2\times d\times|t|)$, respectively. On the other hand, it is quite reasonable to assume d to be a constant because, in general, the set of super-classes has a limited size. Consequently, the IbC algorithm has a time complexity of $O(|t|)$.

Since IbC works in an online mode, it has a low demand on the memory. The primary space overhead introduced by IbC is the hash table H used to save OUSs, which has the same size as the input PET t. Although functions RPEventSet() and OwnerClassSet() also lead to some memory overhead (such as the set of owner classes W and queue Q used to save super-classes), it is within a limited size and can be taken as a constant. Therefore, the space complexity of IbC is $O(|t|)$.

In conclusion, IbC is an effective and efficient OUS-collecting algorithm, which has a linear time and space complexity. Compared with general OUS-collecting approaches (which can collect only general OUSs), IbC can collect general OUSs and RP-OUSs. Although we introduce IbC based on PDA techniques in this study, IbO is a general approach and is also appropriate for PSA.

6 Evaluation

To evaluate our technique, we implemented it in our previous prototype tool ISpecMiner and used the tool to conduct experiments. In this section, we first introduce our dynamic API protocol miner ISpecMiner. Then, we present subjects used in our evaluation. Finally, we conduct several experiments and investigate the following issues:

1. Is our technique effective in terms of mining API protocols?

2. Is the cost (in terms of runtime overhead) of using our technique affordable in practice?

6.1 Prototype tool ISpecMiner

ISpecMiner is a dynamic program specification mining tool developed based on Java 1.6 (Chen et al., 2015a). It leverages the Java agent (Caserta and Zendra, 2014) technique and Javassist (http://en.wikipedia.org/wiki/Javassist; Skala and Petruska, 2014; Tatsubori et al., 2001) to extract OUSs from Java programs dynamically, and then infers class temporal specifications (API protocols). The most distinguishing characteristic of ISpecMiner is that it describes program specifications using a probabilistic model extended from the Markov chain, which has an inherent ability to tolerate noise. Furthermore, since ISpecMiner learns program specifications in an online mode, mined specifications can evolve persistently.

For comparison tests, we implemented both a general OUS-collecting algorithm and our IbC in ISpecMiner. The general OUS-collecting algorithm is similar to IbC shown in Algorithm 1. The difference is that it does not subsume the if-then statement and cannot collect RP-OUSs. For notational convenience, we denote ISpecMiner integrated with the general OUS-collecting algorithm and IbC ISpecMiner-1 and ISpecMiner-2 separately. The latest version of ISpecMiner can be obtained at http://www.ispecminer.com.

6.2 Subjects

Our evaluation is based on two sets of subjects. The first set of subjects (Table 2) is composed of

real-world Java programs. We selected these programs based on the following considerations:

1. Open source software. Though ISpecMiner is a dynamic specification mining tool and the source code is not necessary, it is helpful for us to figure out problems encountered in experiments and validate results.

2. Programs coming from various domains. Programs from various areas may avoid the biases existing in our evaluation.

Table 2 The first set of subjects used for evaluation

Subject	Version	Description	KLoC
FreeMind	0.9	Mind-mapping software	22
RapidMiner	5.3	Environment for machine learning and data mining	513
SQuirreL SQL client	3.4	Java SQL client	253
OpenProj	1.4	Project management software	120

KLoC: kilo lines of code

The second set of subjects (Fig. 8) is an assembly of four orchestrated Java programs, designed to measure the runtime overhead of our technique.

6.3 Investigation of API protocols mined with our technique

To investigate the effect of our method on mining API protocols, we used ISpecMiner-1 and ISpecMiner-2 to mine API protocols from several real-world Java programs separately, and then compared the achieved protocols. The subject programs are shown in Table 2, each run with manual input data. We configured ISpecMiner-1 and ISpecMiner-2 to instrument the classes shown in Table 3 and their super-classes (except for java.lang. Object). The reasons that we selected these classes are: (1) they are commonly used in various kinds of Java programs; (2) they have more than one super-class. Thus, in an ideal case, a general OUS of these classes will derive at least one RP-OUS. Note that due to some technical reasons, we excluded the common super-class java.lang.Object from instrumentation and neglected it when counting the number of super-classes. Take as an example the class java.io.FileInputStream shown at the second row of Table 3. It has two super-classes java.io.InputStream and java.lang.Object. According to our counting method, the number of its super-classes is one.

The statistical results of the API protocols mined by ISpecMiner-1 and ISpecMiner-2 are presented in Table 4. We investigated 19 classes in all, inclusive of the classes listed in Table 3 and their super-classes (in italics). For each class, we presented the number of OUSs and API protocols achieved by ISpecMiner-1 and ISpecMiner-2, respectively. Additionally, the comparison results of the API protocols are given. Overall, ISpecMiner-1 collected 8539 OUSs and

```java
private static void test_LinkedList(){
    Queue<String> queue = new LinkedList<String>();
    String str;
    int i = 0;
    queue.offer("Hello");
    while(i < 100){
        str=queue.poll();
        System.out.print(str);
        queue.offer("hello");
        i++;
    }
    System.out.println();
}
```

(a)

```java
private static void test_Stack(){
    Stack<String> s = new Stack<String>();
    s.push("hello");
    s.push("world");
    s.push("instrumentation technique");
    s.push("This is a test");
    while(!s.empty())
        s.pop();
    s.push("test the time overhead of instrumentation");
    if(!s.isEmpty())
        s.removeAllElements();
}
```

(b)

```java
private static void test_FileReader(){
    FileReader fr;
    BufferedReader br;
    String s;
    try {
        fr = new FileReader("c:\\FileReaderDemo.txt");
        br = new BufferedReader(fr);
        while((s = br.readLine()) != null)
            System.out.println(s);
        fr.close();
    } catch (IOException e) {
        System.out.println(e.getMessage());
        e.printStackTrace();
    }
}
```

(c)

```java
private static void test_FileWriter(){
    String source = "Now is the time for all good men to come
    to the aid of their country and pay their due taxes.";
    String[] srcArray = source.split(" ");
    FileWriter f0;
    int i;
    try{
        f0 = new FileWriter("C:\\FileWriterDemo.txt");
        for(i = 0; i < srcArray.length; i++)
            f0.write(srcArray[i]);
        f0.close();
    }catch(IOException e){
        System.out.println(e.getMessage());
        e.printStackTrace();
    }
}
```

(d)

Fig. 8 The second set of subjects used for evaluation, which subsumes four orchestrated Java programs (a)–(d)

achieved 12 API protocols (we excluded invalid OUSs that had only one constructor method). ISpecMiner-2 collected 25 208 OUSs and achieved 17 API protocols. ISpecMiner-1 and ISpecMiner-2 collected the same number of OUSs for the first 11 classes (rows 1 to 11), and their API protocols are also identical. However, as for their super-classes (rows 12 to 19), ISpecMiner-1 collected few OUSs except for InputStreamReader and OutputStreamWriter. In contrast, ISpecMiner-2 collected many more OUSs than ISpecMiner-1 regarding these classes (except for FilterOutputStream). The overall OUSs collected by ISpecMiner-2 is 1.95 times more than that collected by ISpecMiner-1, which is near the maximum number of folds 1.97 given by Theorem 2. The reason for this huge increase is that ISpecMiner-2 can derive a large number of RP-OUSs for super-classes even if these classes are not used during the run of subject programs. Although ISpecMiner-1 also collected some OUSs regarding the super-classes InputStreamReader and OutputStreamWriter, they are general OUSs and their number is smaller than that collected by ISpecMiner-2. Note that both ISpecMiner-1 and ISpecMiner-2 failed to achieve any OUSs regarding classes DataOutputStream and FilterOutputStream. This is because neither these classes themselves nor their sub-classes were used during the run of subject programs.

Table 3 Investigated classes in the first experiment

No.	Instrumented class	Number of super-classes
1	java.io.PushbackInputStream	2
2	java.io.FileInputStream	1
3	java.io.FileOutputStream	1
4	java.io.BufferedReader	1
5	java.io.BufferedWriter	1
6	java.io.DataInputStream	2
7	java.io.DataOutputStream	2
8	java.io.FileReader	2
9	java.io.FileWriter	2
10	java.io.BufferedInputStream	2
11	java.io.PrintWriter	1

Above all, the API protocols mined by ISpecMiner-2 are more accurate and complete than those mined by ISpecMiner-1. For a close investigation, we present two pairs of API protocols in Fig. 9. As we can see, API protocols (a) and (b) are of the class InputStreamReader mined by ISpecMiner-1 and ISpecMiner-2, respectively. API protocols (c) and (d) are of the class OutputStreamWriter mined by ISpecMiner-1 and ISpecMiner-2, respectively. The API protocols are described using an extended

Table 4 Statistical results of the comparison test on API protocols

No.	Investigated class	OUS number		API protocol*		Comparison**
		ISpecMiner-1	ISpecMiner-2	ISpecMiner-1	ISpecMiner-2	
1	java.io.PushbackInputStream	146	146	√	√	Y
2	java.io.FileInputStream	66	66	√	√	Y
3	java.io.FileOutputStream	26	26	√	√	Y
4	java.io.BufferedReader	46	46	√	√	Y
5	java.io.BufferedWriter	28	28	√	√	Y
6	java.io.DataInputStream	280	280	√	√	Y
7	java.io.DataOutputStream	0	0	×	×	Y
8	java.io.FileReader	21	21	√	√	Y
9	java.io.FileWriter	6	6	√	√	Y
10	java.io.BufferedInputStream	7850	7850	√	√	Y
11	java.io.PrintWriter	31	31	√	√	Y
12	*java.io.FilterInputStream*	0	8192	×	√	N
13	*java.io.InputStreamReader*	26	46	√	√	N
14	*java.io.OutputStreamWriter*	13	19	√	√	N
15	*java.io.FilterOutputStream*	0	0	×	×	Y
16	*java.io.InputStream*	0	8258	×	√	N
17	*java.io.OutputStream*	0	26	×	√	N
18	*java.io.Reader*	0	89	×	√	N
19	*java.io.Writer*	0	78	×	√	N
	Total	8539	25 208	12	17	Y(12), N(7)

* Whether or not an API protocol was achieved; ** whether or not the API protocols mined by ISpecMiner-1 and ISpecMiner-2 are identical

Markov model MCF, where states (rounded rectangles) and transitions (arrows) represent methods and temporal relationships between methods, respectively. An MCF may subsume three kinds of probabilities: initial probability (InitPro), final probability (FinalPro), and transition probability. The initial and final probabilities reflect how probable a method appears at the beginning and at the end of an OUS, respectively. The transition probability is labeled with arrows, indicating how probable a method is deemed before or after another method. Details about MCF can be found in Chen et al. (2015a).

From Fig. 9, we can see that the API protocols mined by ISpecMiner-2 are more complete than those mined by ISpecMiner-1. For example, the API protocol of the class InputStreamReader, mined by ISpecMiner-2 (b), has one more state and four more transitions than those mined by ISpecMiner-1 (a). As for the API protocols of the class OutputStreamWriter shown in (c) and (d), although they have the same number of states, the one mined by ISpecMiner-2 (d) has many more transitions than that mined by ISpecMiner-1 (c). Additionally, by manual inspection, we found that the additional states and transitions were consistent with JDK documentations. Based on the above analysis, we conclude that our technique is helpful for mining complete API protocols.

On the other hand, API protocols mined by ISpecMiner-2 have a more reasonable probability distribution than those mined by ISpecMiner-1: normal behaviors have higher probabilities than abnormal behaviors. Take the API protocol of class InputStreamReader as an example. The final probability (FinalPro) of state close() was increased from 0.6154 (Fig. 9a) to 0.8043 (Fig. 9b), while the final probability of state read(char[], int, int) was decreased

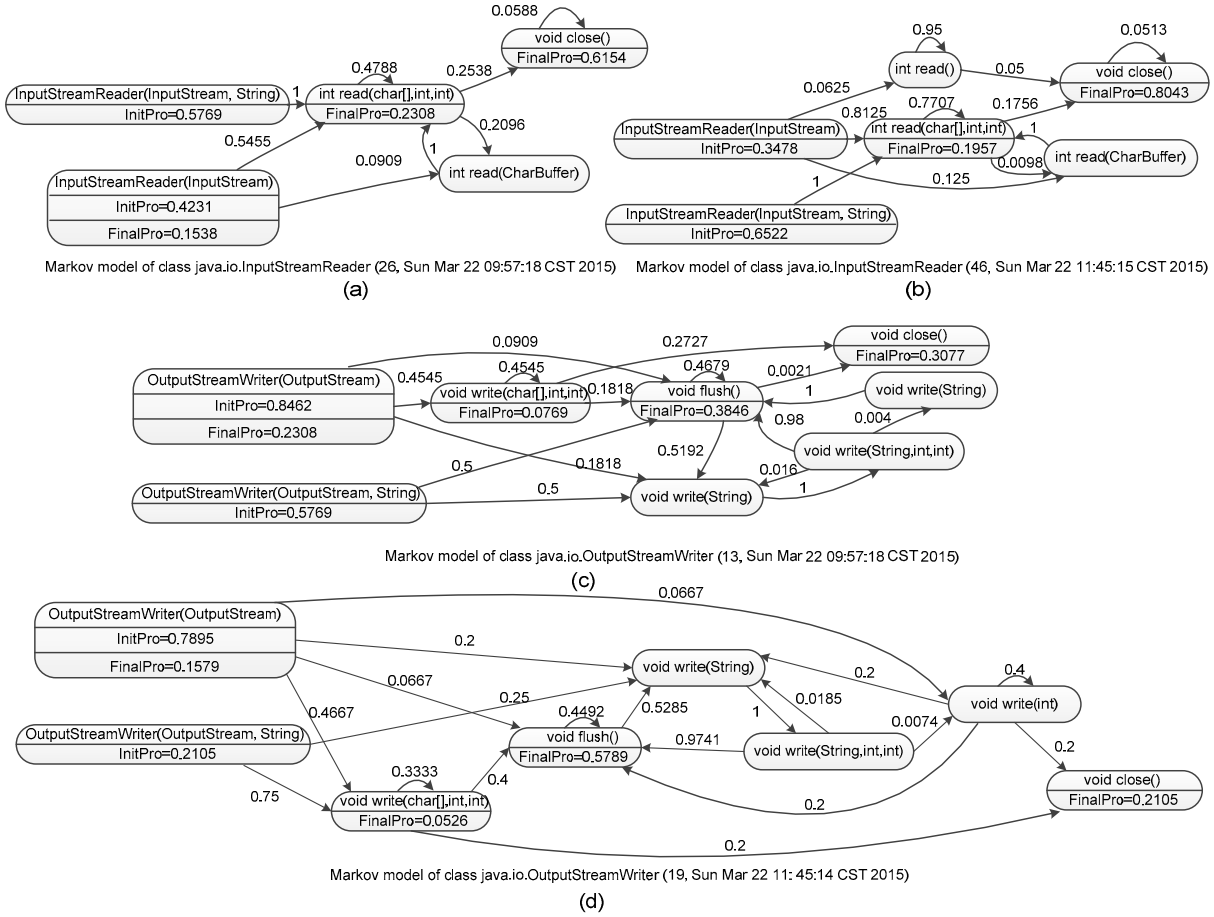

Fig. 9 API protocols achieved in our evaluation: (a) API protocol of InputStreamReader mined by ISpecMiner-1; (b) API protocol of InputStreamReader mined by ISpecMiner-2; (c) API protocol of OutputStreamWriter mined by ISpecMiner-1; (d) API protocol of OutputStreamWriter mined by ISpecMiner-2

from 0.2308 (Fig. 9a) to 0.1957 (Fig. 9b). The probability distribution of the API protocol mined by ISpecMiner-2 is more consistent with JDK documentation: the usage of class InputStreamReader should end with method close(); that is, the API protocol of the class should have only one final state close(). The reason for this is that ISpecMiner-2 collected more OUSs, which can mitigate the problem of overfitting to some extent and achieve more accurate API protocols.

Furthermore, we found that ISpecMiner-2 mined five more API protocols than ISpecMiner-1 regarding the following classes: FilterInputStream, InputStream, OutputStream, Reader, and Writer. Since these classes were not used during the run of subject programs, ISpecMiner-1 hardly collected any OUS regarding them. However, ISpecMiner-2 can derive a large number of RP-OUSs from the general OUSs regarding their sub-classes. It seems that API protocols regarding super-classes are useless for program validation because they are seldom used in application programs. As a matter of fact, there still exist many super-classes that are frequently used in programs, such as InputStreamReader and OutputStreamWriter (it is just because both classes were used during the run of subject programs that ISpecMiner-1 could mine their API protocols). Even if some classes may never be used in programs (such as abstract classes), their API protocols may be useful for program understanding and validation. For example, we can validate the design of an abstract class based on mined protocols, which may be beneficial for validating sub-classes inherited from the abstract class.

In conclusion, we performed a comparison test based on four real-world Java programs and investigated the OUSs and API protocols achieved by ISpecMiner. Experimental results show that IbC can gather 1.95 times more OUSs than the general OUS-collecting algorithm. Additionally, complete and accurate API protocols are more likely to be achieved. Apart from that, our technique can mine API protocols for classes never used in programs, which is beneficial for validating software architectures.

6.4 Investigation of runtime overhead of our technique

In this subsection, we investigate the runtime overhead of our technique, which may threaten the validity of our approach.

Since measuring the execution time of many real-world programs (e.g., programs that require interactions from users during their run) precisely is intractable, we performed the experiment based on a set of contrived Java programs (Fig. 8). As we can see, the subject programs are single-threaded and do not require user interactions. Relying on the subject programs, we conducted the following groups of tests: (1) running each subject program once without instrumentation; (2) running each subject program once with ISpecMiner-1; (3) running each subject program once with ISpecMiner-2. The platform that we used was Intel® Core™ i3-2100 3.1 GHz with 3 GB memory running Windows XP. All groups of tests were based on the same input data and repeated seven times. Additionally, ISpecMiner-1 and ISpecMiner-2 were configured to instrument the following classes: java.util.LinkedList, java.io.BufferedReader, java.util.Stack, java.io.FileReader, and java.io.FileWriter. Note that all the above classes would be used during the run of subject programs. Also, note that program (c) or (d) shown in Fig. 8 takes a file named FileReaderDemo.txt or FileWriterDemo.txt as input, respectively. In our evaluation, FileReaderDemo.txt contains 33 lines of text and FileWriterDemo.txt is empty.

The experimental results are shown in Table 5. As we can see, we present program execution times of the first group (GP1), second group (GP2), and third group (GP3). The runtime overheads of ISpecMiner-1 (ISM-1) and ISpecMiner-2 (ISM-2) and the percentage of increased runtime overhead caused by our technique (Inc) are also given, which are computed based on the following equations:

$$\text{ISM-1} = \text{GP2} - \text{GP1}, \tag{18}$$

$$\text{ISM-2} = \text{GP3} - \text{GP1}, \tag{19}$$

$$\text{Inc} = \frac{\text{ISM-2} - \text{ISM-1}}{\text{ISM-1}} \times 100\%. \tag{20}$$

By averaging the results of all the seven execution times of the tests, we found that the mean program execution time of GP1, GP2, or GP3 was 18.7, 33.9, or 36.3 ms, respectively. The mean runtime overheads of ISpecMiner-1 and ISpecMiner-2 were 15.1 and 17.6 ms, respectively. ISpecMiner-2 caused an average of 21% more runtime overhead than ISpecMiner-1. Since the OUS-collecting algorithms of ISpecMiner-1 and ISpecMiner-2 are nearly the

same except for the functionality of deriving RP-OUSs, the extra runtime overhead is introduced mainly by our IbO technique.

Table 5 Results of runtime overhead investigation

No.	Execution time (ms)			Runtime overhead (ms)		
	GP1	GP2	GP3	ISM-1	ISM-2	Inc (%)
1	17	35	37	18	20	11
2	18	33	36	15	18	20
3	19	33	35	14	16	14
4	25	30	33	5	8	60
5	17	35	37	18	20	11
6	18	38	41	20	23	15
7	17	33	35	16	18	13

From Section 5.2.3, we can see that our IbO algorithm has a time complexity of $O(l)$, where l is the length of input PETs. Thus, although more runtime overhead may be introduced by our technique, it is within an acceptable range. Furthermore, the increased runtime overhead is trivial with respect to the benefit brought by the large number of additional OUSs. Above all, our technique is helpful for improving the accuracy and completeness of mined API protocols, which is a concern more relevant than the runtime overhead for API protocol mining and applications.

6.5 Limitations

In this work, we discussed our technique based on single-object protocols. However, according to the substitution property (Bruce and Wegner, 1986), our technique may be also appropriate for mining multi-object protocols. For example, given the following method call sequence used to mine multi-object protocols regarding objects T_1, T_2, and T_3,

$$< T_1.m_1,\ T_1.m_2,\ T_2.m_3,\ T_1.m_4,\ T_2.m_5,\ T_3.m_6 >, \quad (21)$$

where T_i (i=1, 2, 3) and m_j (j=1, 2, …, 6) denote types and methods, respectively, our technique may derive a set of additional method call sequences:

$$D = \{< T_1'.m_1,\ T_1'.m_2,\ T_2'.m_3,\ T_1'.m_4, T_2'.m_5,\ T_3'.m_6 > \\ |\ T_i' = T_i \vee T_i' \in \mathrm{SUPCLS}(T_i),\ i = 1,2,3\}. \quad (22)$$

The number of derived method call sequences can be computed as follows:

$$|D| = \prod_{i=1}^{3} (\mathrm{SUPCLS}(T_i) + 1) - 1. \quad (23)$$

Let n be the ANoS. We have

$$|D| \approx n^3. \quad (24)$$

Note that compared with single-object protocol mining techniques (in which case $|D| \leq n$), our technique may be more effective in mining multi-object protocols. Above all, the derived method call sequences subsume many more object interactions regarding super-classes, which may lead to more complete multi-object protocols.

Since multi-object protocol mining techniques are quite complex, to make our technique understandable, we confine our work to single-object protocols. Applications of our technique to multi-object protocols are left as an extension of this work.

7 Related work

Many researchers have made significant efforts in mining API protocols. In this section, we introduce some typical studies in this area.

Wasylkowski et al. (2007) mined object usage models (FSA) from Java bytecode and a tool JADET was developed. Lorenzoli et al. (2008) modeled API protocols using EFSM which is an extension from FSM. Alur et al. (2005) synthesized an FSA model of API protocols using L* learning algorithms combined with model checking and abstract interpretation techniques.

Since FSA is a kind of deterministic model with an inability to tolerate noise, many researchers proposed mining API protocols based on probabilistic models. Ammons et al. (2002) proposed to mine temporal specifications among application programming interfaces (API) or abstract data types (ADT) based on probabilistic finite state automaton (PFSA). A PFSA is a nondeterministic finite automaton (NFA), in which each edge is labeled by an abstract interaction and weighted by how often the edge is traversed while generating or accepting scenario strings. To mine temporal specifications, first an off-the-shelf PFSA learner was used to analyze scenario strings and generate a PFSA. Next, another component corer was employed to transform PFSA to NFA by discarding

rarely used edges and weights. The NFA obtained was used for program verification and manual inspection. Chen et al. (2015a) mined class temporal specifications based on an extended Markov model. They first synthesized probabilistic models from a set of OUSs and then transformed the probabilistic models to deterministic models. They discussed the challenge of unconnected models and proposed an approach to select proper threshold values. Based on the computed threshold values, they can eliminate noises and achieve connected models.

Dai et al. (2014) is similar to ours in that both are able to mine API protocols of abstract classes. To obtain the API protocol of an abstract class, they extracted submodels from API protocols of its implementing classes through a state-preserving submodel extraction algorithm. The difference in our work is that it synthesizes the API protocols of abstract classes based on derived RP-OUSs.

Whatever the techniques used, an adequate number of OUSs is essential for mining accurate and complete API protocols. To resolve the problem, existing approaches resort to analyzing a large codebase or more application programs, which require much time overhead. In this study, we proposed an oversampling approach for OUSs, which can collect many times more OUSs than general approaches from a single application program.

8 Conclusions and future work

Different from many existing approaches that increase the number of OUSs by analyzing a large codebase or more programs, we proposed an oversampling approach IbO. IbO is a general approach applicable for object-oriented programs. Additionally, it can be integrated with both PSA and PDA techniques. Based on the inheritance relationship among classes, IbO can collect n times more OUSs than general approaches, where n is the ANoS of all general OUSs. To ease the use of our technique, we integrated IbO with PDA and proposed an OUS-collecting algorithm IbC, which can collect general OUSs and RP-OUSs.

Based on our previous prototype ISpecMiner, we investigated the effect of our method on mining API protocols and the runtime overhead incurred by our approach. Experimental results showed that our technique collected 1.95 times more OUSs than a general approach, and that accurate and complete API protocols were more likely to be achieved. Although many API protocols achieved by our technique were of super-classes (which may be seldom or even never used in programs), they are beneficial for program understanding and validation. What may threaten the validity of our technique is the significant runtime overhead. However, through formal analysis and experimental survey, we found that the runtime overhead caused by our technique was within an acceptable range. Above all, it is trivial with respect to the benefit brought by the large number of additional OUSs collected by our technique.

For simplicity, in this paper we discussed our technique based on single-object protocols. However, as analyzed in Section 6.5, our technique may also be applicable for multi-object protocols. We leave the exploration of this to future work. To further evaluate the effectiveness of our technique, we will perform extensive experiments on the large-scale and popular APIs.

References

Alur R, Černý P, Madhusudan P, et al., 2005. Synthesis of interface specifications for Java classes. *ACM SIGPLAN Not*, 40(1):98-109. https://doi.org/10.1145/1047659.1040314

Ammons G, Bodík R, Larus JR, 2002. Mining specifications. *ACM SIGPLAN Not*, 37(1):4-16. https://doi.org/10.1145/565816.503275

Bruce KB, Wegner P, 1986. An algebraic model of sybtypes in object-oriented languages (draft). *ACM SIGPLAN Not*, 21(10):163-172. https://doi.org/10.1145/323648.323756

Caserta P, Zendra O, 2014. JBInsTrace: a tracer of Java and JRE classes at basic-block granularity by dynamically instrumenting bytecode. *Sci Comput Program*, 79:116-125. https://doi.org/10.1016/j.scico.2012.02.004

Chang RY, Podgurski A, Yang J, 2007. Finding what's not there: a new approach to revealing neglected conditions in software. Proc Int Symp on Software Testing and Analysis, p.163-173. https://doi.org/10.1145/1273463.1273486

Chen D, Huang RB, Qu BB, et al., 2014. Improving static analysis performance using rule-filtering technique. Proc 26th Int Conf on Software Engineering and Knowledge Engineering, p.19-24.

Chen D, Huang RB, Qu BB, et al., 2015a. Mining class temporal specification dynamically based on extended Markov model. *Int J Softw Eng Knowl Eng*, 25(3):573-604. https://doi.org/10.1142/S0218194015500047

Chen D, Zhang YD, Wang RC, et al., 2015b. Extracting more object usage scenarios for API protocol mining. Proc 27th Int Conf on Software Engineering and Knowledge

Engineering, p.607-612. https://doi.org/10.18293/SEKE2015-212

Chen D, Zhang YD, Wei W, et al., 2017. Efficient vulnerability detection based on an optimized rule-checking static analysis technique. *Front Inform Technol Electron Eng*, 18(3):332-345. https://doi.org/10.1631/FITEE.1500379

Cook JE, Wolf AL, 1998. Discovering models of software processes from event-based data. *ACM Trans Softw Eng Methodol*, 7(3):215-249. https://doi.org/10.1145/287000.287001

Dai ZY, Mao XG, Lei Y, et al., 2014. Compositional mining of multiple object API protocols through state abstraction. *Sci World J*, Article 171 647.

Dallmeier V, Lindig C, Wasylkowski A, et al., 2006. Mining object behavior with ADABU. Proc Int Workshop on Dynamic Systems Analysis, p.17-24. https://doi.org/10.1145/1138912.1138918

Dallmeier V, Knopp N, Mallon C, et al., 2012. Automatically generating test cases for specification mining. *IEEE Trans Softw Eng*, 38(2):243-257. https://doi.org/10.1109/TSE.2011.105

Engler D, Chen DY, Hallem S, et al., 2001. Bugs as deviant behavior: a general approach to inferring errors in systems code. *ACM SIGOPS Oper Syst Rev*, 35(5):57-72. https://doi.org/10.1145/502059.502041

Ernst MD, Perkins JH, Guo PJ, et al., 2007. The Daikon system for dynamic detection of likely invariants. *Sci Comput Program*, 69(1-3):35-45. https://doi.org/10.1016/j.scico.2007.01.015

Kernighan BW, Ritchie DM, 1988. The C Programming Language (2nd Ed.). Prentice Hall, Englewood Cliffs, NJ.

Li ZM, Zhou YY, 2005. PR-miner: automatically extracting implicit programming rules and detecting violations in large software codes. *ACM SIGSOFT Softw Eng Not*, 30(5):306-315. https://doi.org/10.1145/1095430.1081755

Liskov B, 1988. Keynote address—data abstraction and hierarchy. *ACM SIGPLAN Not*, 23(5):17-34. https://doi.org/10.1145/62139.62141

Lorenzoli D, Mariani L, Pezzè M, 2008. Automatic generation of software behavioral models. Proc 30th Int Conf on Software Engineering, p.501-510. https://doi.org/10.1145/1368088.1368157

Pradel M, Gross TR, 2012. Leveraging test generation and specification mining for automated bug detection without false positives. Proc 34th Int Conf on Software Engineering, p.288-298. https://doi.org/10.1109/ICSE.2012.6227185

Pradel M, Jaspan C, Aldrich J, et al., 2012. Statically checking API protocol conformance with mined multi-object specifications. Proc 34th Int Conf on Software Engineering, p.925-935. https://doi.org/10.1109/ICSE.2012.6227127

Ramanathan MK, Grama A, Jagannathan S, 2007. Static specification inference using predicate mining. *ACM SIGPLAN Not*, 42(6):123-134. https://doi.org/10.1145/1273442.1250749

Shoham S, Yahav E, Fink S, et al., 2007. Static specification mining using automata-based abstractions. Proc Int Symp on Software Testing and Analysis, p.174-184. https://doi.org/10.1145/1273463.1273487

Skala V, Petruska R, 2014. A new approach to hash function construction for textual data: a comparison. Proc 4th World Congress on Information and Communication Technologies, p.39-44. https://doi.org/10.1109/WICT.2014.7077299

Tatsubori M, Sasaki T, Chiba S, et al., 2001. A bytecode translator for distributed execution of "legacy" Java software. Proc 15th European Conf on Object-Oriented Programming, p.236-255.

Thummalapenta S, Xie T, 2011. Alattin: mining alternative patterns for defect detection. *Autom Softw Eng*, 18(3-4): 293-323. https://doi.org/10.1007/s10515-011-0086-z

Wasylkowski A, 2007. Mining object usage models. Proc 29th Int Conf on Software Engineering, p.93-94. https://doi.org/10.1109/ICSECOMPANION.2007.49

Wasylkowski A, Zeller A, Lindig C, 2007. Detecting object usage anomalies. Proc 6th Joint Meeting of the European Software Engineering Conf and the ACM SIGSOFT Symp on Foundations of Software Engineering, p.35-44. https://doi.org/10.1145/1287624.1287632