Extracting More Object Usage Scenarios for API Protocol Mining

Deng Chen[1, a], Yanduo Zhang[2, b], Rongcun Wang[3, c], Binbin Qu[4, d], Jianping Ju[5, e], Wei Wei[1, f]

[1] Industrial Robot Engineering Center, Wuhan Institute of Technology, Wuhan, P.R. China

[2] Hubei Provincial Key Laboratory of Intelligent Robot, Wuhan Institute of Technology, Wuhan, P.R. China

[3] School of Computer Science and Technology, China University of Mining and Technology, Xuzhou, P.R. China

[4] School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, P.R. China

[5] School of Electronic Information, Wuhan University, Wuhan, P.R. China

[a] chendeng8899@hust.edu.cn

[b] zhangyanduo@hotmail.com

[c] rcwang@hust.edu.cn

[d] bbqu@hust.edu.cn

[e] gjdxjjp@whu.edu.cn

[f] weiwei@huawei-elec.com

Abstract—**Automatic protocol mining is a promising approach to infer precise and complete API protocols. However, the effect of the approach largely depends upon the quality of input object usage scenarios, in terms of noise and diversity. This paper aims to extract as many object usage scenarios as possible from object-oriented programs for automatic protocol mining. A large corpus of object usage scenarios can help with eliminating noise accurately and is likely to be diverse. Therefore, precise and complete protocols may be achieved. Given an object-oriented program *p*, generally, object usage scenarios that can be collected from a run of *p* is not more than the number of instances used in *p*. Relying on the inheritance relationship among classes, our technique can extract a maximum of *n* times more object usage scenarios from *p*, where *n* is the average inheritance depth of all object usage scenarios in *p*. In order to investigate the effect of our technique on mining protocols, we implement it in our previous prototype tool ISpecMiner and use the tool to mine protocols from several real-world applications. The experimental results show that our technique is promising to achieve complete and precise API protocols. In addition, protocols of classes that have not been used in programs can be also achieved, which is helpful for program documentation and understanding.**

Keywords-object usage scenario; mining API protocol; object-oriented program; program verification

I. INTRODUCTION

Many application programming interfaces (APIs) impose protocols, that is, temporal constraints regarding the order of calls of API methods. 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`. API clients that violate such protocols do not obtain the desired behaviors and may even crash the program [1].

(DOI reference number: 10.18293/SEKE2015-212)

Automatic protocol mining [2]-[3] is a promising approach to infer precise and complete API protocols. These approaches first extract object usage scenarios from program applications statically or dynamically. Then, they take object usage scenarios as input and synthesize protocols based on sequential data mining techniques. However, the effect of these approaches largely depends upon the quality of input object usage scenarios: (1) noisy object usage scenarios will incur imprecision to mined protocols; and (2) in order to mine complete protocols, a set of diverse object usage scenarios is required.

Instead of improving the quality of input object usage scenarios directly, our work aims to extract as many object usage scenarios as possible from object-oriented programs for automatic protocol mining. Since a large corpus of object usage scenarios can compensate the inaccuracy caused by noise and is likely to be diverse [4], precise and complete protocols may be achieved. Generally, the number of object usage scenarios that can be collected from a run of an object-oriented program is less than or equal to the number of instances used in the program. Therefore, if a class is seldom used in a program, we will achieve insufficient object usage scenarios. Although feeding protocol miners more programs can mitigate the problem to some extent, much time overhead will be incurred.

Our technique is based on the following heuristic for object-oriented programs. Let c_1 and c_2 be two classes. If c_2 inherits from c_1, c_2 will inherit the set of public methods (we omit other kinds of methods, e.g. protected methods and private methods, because API protocols always consider public methods of classes) M of c_1 as well as the temporal constraints regarding the order of calls of methods

in *M*. Or in other words, c_2 should not violate the temporal constraints regarding the order of calls of methods in *M* imposed by c_1 even if it overrides the inherited methods. Consequently, given an object usage scenario *u* of class c_2, *u* should comply with the API protocols of c_2 as well as that of c_1. Based on the above analysis, we derive an extra object usage scenario u' from *u*, which consists of methods inherited from c_1. The extra object usage scenario is used to synthesize protocols of class c_1. Theoretically, given an object-oriented program *p*, our technique can maximally extract *n* times more object usage scenarios from *p* than general approaches, where *n* is the average inheritance depth of all object usage scenarios in *p*. To investigate our technique's feasibility and effectiveness, we implemented it in our previous dynamic program specification mining tool ISpecMiner and used the tool to conduct experiments. Results of the experiments show that our technique is promising to achieve complete and precise protocols.

The contributions of this paper are:

- A technique that can extract more object usage scenarios from object-oriented programs than existing approaches.
- A formal discussion about how many more object usage scenarios can be collected by our technique.
- Investigation of the effect of our technique on mining API protocols.

II. PRELIMINARY

In this section, we present some preliminaries that our work is based on.

DEFINITION 1 (Object Usage Scenario). Let *c* be a class. An *Object Usage Scenario* (*OUS*) of *c* is a method call sequence, all methods in the sequence are called on a same instance of *c*. Assume that *s* is an instance of class *c*. We use *ous*(*s*) to denote an object usage scenario of *c*, each method of which is called on *s*. Furthermore, we use *OUS*(*c*) to denote the set of object usage scenarios of class *c*, each element of which is an object usage scenario of an instance of *c*.

Consider the Java program illustrated in Figure 1, which makes use of classes FileInputStream and FileOutputStream. Let's assume that the loop iterates only once. We will achieve the following OUSs regarding

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

Figure 1. Java program example.

```
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)

Figure 2. Program examples of inheritance relationship.

instance fis and fos respectively:

- *ous*(fis): <FileInputStream(), read(), close()>
- ous(fos): <FileOutputStream(), write(), close()>

As we can see, an OUS *u* of class *c* represents a use case about how client programs should use methods of *c*. Therefore, through extracting common patterns from a set of OUSs of *c*, we can infer protocols of class *c*. Additionally, given an object-oriented program *p*, OUSs that can be collected from a run of *p* is less than or equal to the number of instances used in *p*. In Section 3, we will show that our technique can extract multiple times more OUSs than general approaches.

III. OUR TECHNIQUE

In object-oriented programs, classes have inheritance relationships among them. Consider the Java programs illustrated in Figure 2, 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 subclass will inherit public methods of its superclasses, the above classes have the public methods listed in Table I. Our approach is based on the following heuristic.

HEURISTIC 1. Let *r* be a class. *M* is the set of public methods of *r*. We use ρ_r^M to denote the API protocol regarding methods in *M* imposed by class *r*. Assume that *c* is a subclass of *r*, which should inherit all methods in *M* from *r*. We have $\rho_c^M \circ \rho_r^M$, where $\circ$ represents that

TABLE I. PUBLIC METHODS OF CLASSES WITH INHERITANCE RELATIONSHIPS. METHODS INHERITED FROM SUPERCLASSES ARE IN ITALIC.

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

protocol ρ_c^M is *equivalent to or stricter than* ρ_r^M. Or in other words, the implementation of a subclass should not violate API protocol imposed by its superclasses.

We make the heuristic based on the following literature: From the perspective of data abstraction and hierarchy [5], a *subtype* is one whose objects provide all the behavior of objects of another type (the *supertype*) plus something extra. Furthermore, we have the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T [6]. 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 reimplemented in a subclass, the exceptions thrown in superclasses should be inherited. For example, assume that E is the set of exceptions thrown by method m01 defined in class A0 (shown in Figure 2). When we overwrite method m01 in class A1, all exceptions in E should also be thrown, that is, a subclass should not violate restrictions on exceptions imposed by its superclasses. Our case is similar to the exception handling mechanism in Java language. What is different is that, we consider constraints regarding order of calls of methods rather than exceptions. Whatever, according to the data abstraction and hierarchy principle, we have the following conclusion: a subtype should not violate the API protocol imposed by its supertype regarding inherited methods, otherwise the substitution property cannot be satisfied. What may occur is that the subtype has a stricter restriction than supertypes on the order of calls of inherited methods.

According to Heuristic 1, given an OUS u of class c, if u can be accepted by the API protocol of c, u should also comply with the protocol of the superclasses of c regarding inherited methods, because the protocol imposed by c is equivalent to or stricter than that imposed by superclasses. Consequently, we can derive an extra OUS from u, which consists of calls of inherited methods. We call the derived OUS *inherited sub-OUS*, which can be used to mine protocols of the superclasses of c. Formally, we give the following definitions.

DEFINITION 2 (Sub-OUS). Given an OUS u, OUS u' is a *sub-OUS* of u, if it satisfies the following requirements.

- Each method call in u' is included in u.
- Let m_1 and m_2 be two method calls in u', the temporal relationship between m_1 and m_2 should be consistent in u' and u, that is, if m_1 *precedes* m_2 in u', m_1 should also *precede* m_2 in u.

DEFINITION 3 (Inherited Sub-OUS). Given an OUS u of class c which consists of calls of inherited methods and those defined in c itself, u' is a sub-OUS of u, each element of which is a call of method inherited from a superclass c' of c, we call u' an *inherited sub-OUS* of u from c' and denote it by $isub-OUS(u, c')$.

For example, given an OUS u: <m01, m11, m03, m02, m12, m31, m32, m20, m23, m33> of class A3, according to Definition 3, we can derive the following inherited sub-OUSs.

- *isub-OUS*(u, A0): <m01, m03, m02>
- *isub-OUS*(u, A1): <m01, m11, m03, m02, m12>
- *isub-OUS*(u, A2): <m01, m11, m03, m02, m12, m20, m23>

As we can see, the above inherited sub-OUSs are sub-OUS of u. In addition, they consist of methods inherited from superclass A0, A1 and A2 respectively. Based on Heuristic 1, the inherited sub-OUSs should satisfy the API protocols of class A0, A1 and A2 respectively. Consequently, aside from OUS u which can be used to mine protocol of class A3, we will achieve three additional OUSs.

In addition, given an OUS u of class c, the number of inherited sub-OUSs of u is less than or equal to the inheritance depth of c. On the other hand, each instance in an object-oriented program will generate an OUS. Therefore, given an object-oriented program, the extra OUS (inherited sub-OUS) collected by our technique is maximally n times the number of instances defined in the program, where n is the average inheritance depth of all OUSs (excluding inherited sub-OUSs) in the program.

IV. FORMAL ANALYSIS OF THE EFFECT OF OUR TECHNIQUE

In this subsection, we analyze the effect of our technique formally.

Let's assume that p is a Java program. It subsumes the following OUSs u_1, u_2, …, u_m, which have an inheritance depth of d_1, d_2, …, d_m, respectively. Given an OUS u of class c, we call inheritance depth of c the inheritance depth of u. It must be noted that the number of inherited sub-OUSs of u is less than or equal to its inheritance depth. Therefore, the maximum total number of additional OUSs (inherited sub-OUSs) that can be collected by our technique from p is $d_1 + d_2 + \ldots + d_m$. Let $\bar{d}$ be the average inheritance depth of all OUSs (excluding inherited sub-OUSs) included in p. We have $d_1 + d_2 + \cdots + d_m = m \times \bar{d}$. Based on the above analysis, we reach the conclusion that our technique can extract maximally n times more OUSs from an object-oriented program, where n is the average inheritance depth of OUSs (excluding inherited sub-OUSs) in the program.

TABLE II. SUBJECT PROGRAMS. KLOC: THOUSANDS OF LINES OF CODE.

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

V. EXPERIMENTS

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 ISpecMiner. Then, we present subjects that are used in our evaluation. Finally, we compared API protocols achieved under our technique and general approaches.

A. Prototype Tool ISpecMiner

ISpecMiner [7] is a dynamic program specification mining tool developed based on Java 1.6. It leverages Java agent [8] technique as well as Javassist [9], [10] to extract OUSs from Java application 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 Markov chain. Probabilistic models have an inherent ability to tolerate noises. Furthermore, since ISpecMiner learns program specifications in an online mode, mined specifications can be evolved persistently. As a result, more universal program specifications can be achieved.

The number of OUSs that ISpecMiner extracts from a Java program is near the number of instances of classes used in the program. In this work, to prepare ISpecMiner for our experiments, we implemented our technique in it. In the remainder of this section, we denote original ISpecMiner and ISpecMiner with our technique by ISpecMiner-Ⅰ and ISpecMiner-Ⅱ respectively. The latest version of ISpecMiner can be obtained at the URL http://www.ispecminer.com.

B. Subjects

The subjects used in our experiment are shown in Table II, which are real-world Java programs. These programs are selected based on the following criteria:

- Open source software. Though ISpecMiner is a dynamic specification mining tool and source code is not necessary, it is helpful for us to figure out problems encountered in experiments and validate results.
- Large-scaled software. Large-scaled software contains a large number of OUSs, which is helpful for our comparison test.
- Applications coming from various domains. Applications from various areas may avoid the biases introduced in our experiments.

TABLE III. INSTRUMENTED CLASSES

	Instrumented Class	Inheritance Depth
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

C. Investigation of API protocols

In order to investigate the effect of our method, we used ISpecMiner-Ⅰ and ISpecMiner-Ⅱ to mine API protocols from several real-world Java programs respectively, and then compared the achieved protocols. The subject programs are shown in Table II, each of which was run with manual input data. We configured ISpecMiner-Ⅰ and ISpecMiner-Ⅱ to instrument classes illustrated in Table III and their superclasses. The reasons we selected these classes are as follows: (1) they are commonly used in various kinds of Java programs; and (2) they have an inheritance depth greater than zero. Thus, an OUS of these classes will derive at least one inherited sub-OUS. It is worth noting that we exclude the common superclass java.lang.Object of all classes in Java when counting inheritance depth. Take for example, class java.io.FileInputStream, which has two superclasses java.io.InputStream and java.lang.Object. According to our counting method, it has an inheritance depth of one.

Experimental results are illustrated in Figure 3. We present the API protocols of class InputStreamReader, InputStreamWriter and FilterInputStream sequentially from the first row to the last row. At each row, the left protocol is mined by ISpecMiner-Ⅰ and the right one is mined by ISpecMiner-Ⅱ. The protocols are described using an extended Markov model MCF, where states and transitions represent methods and temporal relationships between methods respectively. Details about MCF please refer to [7]. From Figure 3, we can see that protocols in the right column have more states or transitions than those in the left column. For example, the API protocol of class InputStreamReader mined by ISpecMiner-Ⅱ has one more state and four more transitions than that mined by

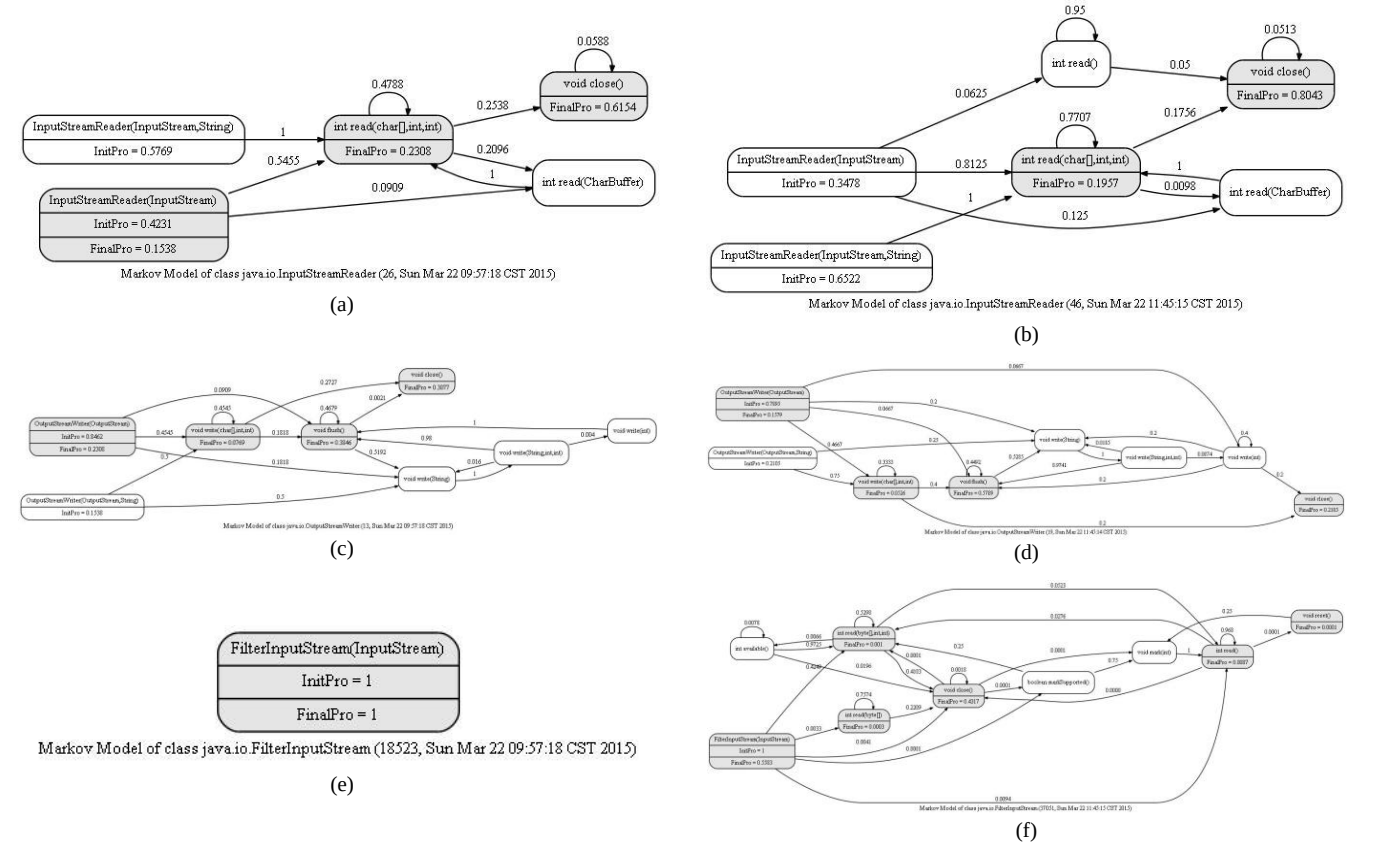

Figure 3. Part of API protocols mined in our experiment.

ISpecMiner-Ⅰ. As to the protocols of class OutputStreamWriter shown in the second row, although they have the same number of states, the right one has many more transitions than the left one. By manual inspection, we found that the additional states and transitions are consistent with JDK documentations. Since the number of states and transitions reflects the comprehensiveness of protocols to some extent, we have the conclusion that our technique is helpful for mining comprehensive protocols.

On the other hand, our approach can impact the probabilities attached with states and transitions: probabilities of normal behaviors will be enhanced and that of abnormal behaviors will be suppressed. For example, the final probability (FinalPro) of state close() in protocol of class InputStreamReader shown in the first row is increased from 0.6154 to 0.8043 and that of state read(char[],int,int) is decreased from 0.2308 to 0.1957. Since the difference between normal and abnormal behaviors is enlarged, noisy states and transitions can be eliminated accurately when transforming probabilistic models to deterministic models using a probability threshold.

Additionally, our technique can achieve protocols that cannot be mined by general approaches. Take for example, the superclass FilterInputStream of DataInputStream and BufferedInputStream. Since the class has not been covered during the run of subject programs, a protocol shown in Figure 3 (e) with only constructor method was achieved by ISpecMiner-Ⅰ (the constructor method of class FilterInputStream may be called by constructor method of its subclasses). In contrast, ISpecMiner-Ⅱ generated a more complete protocol illustrated in Figure 3 (f), because our method can derive inherited sub-OUSs. It seems that protocols as FilterInputStream are useless for program validation, because they are seldom used in application programs. However, there still exist many superclasses which are frequently used in programs, such as InputStreamReader and OutputStreamWriter. Since they have been used in subject programs, we can achieve relative complete protocols based on general approaches as shown in Figure 3 (a) and (c). Even if some classes will be never used in programs (such as abstract classes), their protocols may be useful in program documentation and understanding. For example, we can validate the design of an abstract class based on mined protocols.

D. Related Work

Many researchers have paid significant efforts in mining API protocols. For instance, Wasylkowski et al. [11] proposed to mine object usage models from Java bytecode

and a tool JADET was developed. Lorenzoli et al. [15] modeled API protocols using EFSM which extends from FSM. Alur et al. [16] synthesized 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 inability to tolerate noise, many researchers proposed to mine API protocols based on probabilistic models. For example, Ammons et al. [17] proposed to mine protocols among application programming interfaces (API) or abstract data types (ADT) based on probabilistic finite state automaton (PFSA). Chen et al. [7] proposed to mine class temporal specifications based on an extended Markov model. Whatever techniques, the quality of input OUSs is important for mining precise and complete protocols. However, little attention has been paid in this area. In this paper, we proposed an approach to collect as many OUSs as possible for automatic protocol mining. A large repository of OUSs can complement the inaccuracy caused by noises and is likely to be diverse. Currently, a common approach to collect more OUSs is feeding protocol miners more application programs, which will incur significant time overhead. Different from that, our technique can extract more OUSs from a single application program.

VI. CONCLUSIONS

Automatic protocol mining is a promising approach to infer precise and complete API protocols. Many researchers have paid significant efforts in this area. However, little attention has been paid on collecting high quality OUSs. In this paper, we proposed an approach to collect more OUSs for API protocol mining. Our technique is based on the inheritance relationship among classes. Given an object-oriented program *p*, theoretically, *n* times more OUSs can be extracted by our technique from *p* than general approaches, where *n* is the average inheritance depth of all OUSs in *p*. In the Experimental Section, we investigated the effect of our approach on mined API protocols and found that our technique is promising to achieve complete and precise protocols. Additionally, our technique can mine protocols even if the corresponding classes have not been covered during the run of application programs. Although these protocols may be useless for program validation, they can be used for program documentation and understanding.

ACKNOWLEDGMENT

Supported by Natural Science Foundation of Hubei Province (No. 2014CFB1006).

REFERENCES

[1] Pradel, M. and Gross, T. R. Leveraging test generation and specification mining for automated bug detection without false positives. In *ICSE'12: Proceedings of the 34th International Conference on Software Engineering*. Zurich, Switzerland, 2012, 288-298.

[2] Ramanathan, M. K., Grama, A., et al. Static specification inference using predicate mining. *SIGPLAN Not.* 2007, 42(6), 123-134.

[3] Shoham, S., Eran, Y., et al. Static specification mining using automata-based abstractions. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. United Kingdom: ACM, London, 2007.

[4] Engler, D., Chen, D., et al. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.* 2001, 35(5), 57-72.

[5] Liskov, B. Data abstraction and hierarchy. *SIGPLAN Not.* 1987, 23(5), 17-34.

[6] Bruce, K.B. and Wegner, P. An algebraic model of sybtypes in object-oriented languages. SIGPLAN Not. 1986, 21(10), 163-172.

[7] Chen, D., Huang, R., et al. Ming class temporal specification dynamically based on extended Markov model. *International Journal of Software Engineering and Knowledge Engineering*. 2013, in press.

[8] Caserta, P. and Zendra, O. JBInsTrace: a tracer of Java and JRE classes at basic-block granularity by dynamically instrumenting bytecode. *Science of Computer Programming*, 2014, 79 (SI), 116-125.

[9] Tatsubori, M., Sasaki, T., et al. A bytecode translator for distributed execution of "legacy" Java software. In *Proceedings of the 15th European Conference on Object-Oriented Programming*. Springer-Verlag, 2001.

[10] Javassist, 2013. http://en.wikipedia.org/wiki/Javassist.

[11] Wasylkowski, A. Mining object usage models. In *Companion to the Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 2007.

[12] Wasylkowski, A., Zeller, A., et al. Detecting object usage anomalies. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Croatia: ACM, Dubrovnik, 2007.

[13] JADET, 2014. http://www.st.cs.uni-saarland.de/models/jadet/.

[14] Dallmeier, V., Lindig, C., et al. Mining object behavior with ADABU. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*. China: ACM, Shanghai, 2006.

[15] Lorenzoli, D., Mariani, L., et al. Automatic generation of software behavioral models. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*. Germany: ACM, Leipzig, 2008.

[16] Alur, R., C˘erny, P., et al. Synthesis of interface specifications for Java classes. *SIGPLAN Not*. 2005, 40 (1), 98-109.

[17] Ammons, G. and Bodik, R., et al. Mining specifications. *SIGPLAN Not.* 2002, 37 (1), 4-16.