Mining Universal Specification Based on Probabilistic Model

Deng Chen[1, a], Yanduo Zhang[2, b], Rongcun Wang[3, c], Xun Li[2, d], Li Peng[4, 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] Hubei Radio & TV University, Wuhan, P.R. China
[a] chendeng8899@hust.edu.cn
[b] zhangyanduo@hotmail.com
[c] rcwang@hust.edu.cn
[d] linuxfly@gmail.com
[e] peling9901@126.com
[f] weiwei@huawei-elec.com

***Abstract*—Class temporal specification is a kind of important program specifications, which specifies that methods of a class should be called in a particular sequence. Dynamic specification mining is a promising approach to achieve this kind of specifications automatically. However, they always infer partial specifications, that is, the mined specifications are biased to input programs or program execution traces. In this paper, we propose to mine class temporal specifications based on a probabilistic model in an online mode. Since our method can evolve mined specifications persistently, universal specifications can be achieved. To investigate our technique's feasibility and effectiveness, we implemented it in a prototype tool ISpecMiner and used the tool to perform experiments. Experimental results show that our method is promising to infer universal specifications if sufficient traces are provided for mining.**

Keywords- program specification mining; Markov model; class temporal specification; dynamic analysis; program execution trace

I. INTRODUCTION

Class temporal specification (which is also referred to as component interface [1], object behavior model [2], object usage model [3], [4], etc.) is an important kind of program specifications, which imposes temporal constraints regarding the order of calls of class public methods (a public method of class *c* is a method that can be accessed outside *c*). 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 specifications do not obtain the desired behavior and may even crash the program [5]. However, class temporal specification is always implicit in programs and undocumented. Even when available, there is no guarantee of their consistence, completeness, and correctness. Dynamic specification mining is a promising approach to resolve the problem.

Dynamic specification mining techniques [6]-[8] run applications with test cases generated automatically or manually, and extract specifications from program execution traces. Since dynamic specification mining techniques do not require program source code as input, compared with static specification mining [9]-[11], they can be used extensively, especially when source code is unavailable. However, existing dynamic specification miners (such as `ADABU` [2]) always achieve partial specifications. In order to mine specifications, these miners first run an application program and collect program execution traces into a traces file leveraging instrumentation techniques. Then, they take the trace file as input and synthesize specifications based on various kinds of sequential data mining approaches. Each run of an application will generate a trace file and corresponding specifications. The problem with this approach is that mined specifications may be biased to the application program and input traces.

In this paper, we propose to mine class temporal specifications in an online mode. Different from existing work, our approach does not save program execution traces in any file. It takes each method call from an execution trace sequentially and evolves existing specifications or creates a new one. The online approach does not require loading all traces into memory at once. Thus, it has minimum space overhead. Additionally, since execution traces extracted from different application programs can be used to refine existing specifications persistently, universal specifications may be achieved. Another characteristic of our technique is that, we describe class temporal specification using a probabilistic model extended from Markov chain. Compared with commonly used Finite State Automaton (FSA), probabilistic model has an inherent ability to tolerate noise. Above all, it can facilitate our online mining strategy.

To investigate the effect of our approach, we implemented our technique in a prototype tool ISpecMiner and used it to conduct experiments. Experimental results show that, our approach is promising to achieve universal specifications, if enough application programs are provided for learning.

The contributions of this paper are as follows:

(DOI reference number: 10.18293/SEKE2015-219)

- An online approach is used to mine class temporal specifications.
- A probabilistic model extended from Markov chain is used to describe class temporal specifications.
- A prototype tool `ISpecMiner` that implements our technique is presented.
- Experiments are performed to investigate the effect of our method.

The rest of this paper is organized as follows: Section II discusses related work. Section III introduces our technique. Section IV presents our experimental results. Section V gives our conclusions.

II. Related Work

Generally, program specification mining techniques can be categorized into static analysis approaches and dynamic analysis approaches. These two kinds of approaches collect method call sequences (or program execution traces) in different manners. Static analysis approaches do not require running application programs. They extract method call sequences from program source code, bytecode or other artifacts based on program static analysis techniques [12]. Dynamic analysis approaches do not take program source code as input. They collect program execution traces by running instrumented application programs. After that, temporal specifications can be synthesized from method call sequences (or program execution traces) based on sequential data mining techniques.

Currently, a commonly used mining approach is based on FSA. For instance, `Wasylkowski` et al. [1] proposed to mine object usage models (which are finite state automata) from Java bytecode and a tool `JADET` was developed. `Lorenzoli` et al. [13] modeled class temporal specification using EFSM which extends from FSM. `Alur` et al. [14] synthesized FSA model of class temporal specification using L* learning algorithms combined with model checking and abstract interpretation techniques. These approaches work in a similar manner. First, they split program execution traces into a set of object usage scenarios (an object usage scenario is a method call sequence, all method calls of which have the same receiver object). Then, they reduce the problem of inferring temporal specifications from a set of method call sequences (or traces) to the well known grammar inference problem [15] by regarding method call sequences and specifications as sentences and languages respectively. As a result, a specification is described using one or multiple finite state automata, where states represent states of involved objects and transitions represent method calls. Method calls in each path from an initial state to a final state constitute a valid execution trace. Figure 2 shows an example of specification for class `java.io.FileOutputStream`. The specification illustrates that, to use class `FileOutputStream`, we should first initiate it through calling its constructor method. Next, we can call method `write(byte[],int,int)` multiple times to write data into the stream. Finally, method `close()` should be called to close the stream.

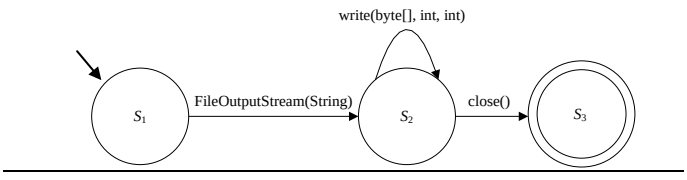

Figure 1. Temporal specification of class `FileOutputStream` described using FSA.

FSA is a kind of deterministic model with inability to tolerate noise. `Ammons` et al. [16] 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 generated 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.

However, existing tools (such as `Daikon` [17] and `ADABU` [2]) always work in a two-step mode. In the first step, they collect execution traces from application programs using a tracer and then store the traces in a trace file. In the next step, they take the trace file as input and synthesize specifications. Each run of an application will generate a trace file and corresponding specifications. The problem with this approach is that results of multiple runs cannot be merged. Thus, the mined specifications are biased to the input trace file. In this work, we mine class temporal specifications based on an online approach. In addition, a probabilistic model extended from Markov chain is employed to describe specifications.

III. Our Technique

In this section, we present our online specification mining technique. We first provide an intuitive description of our technique and then discuss its main characteristics in detail.

A. General Approach

The working principle of our approach is illustrated in Figure 2. The `tracer` is responsible for collecting program execution traces from application programs via instrumentation technique. Different from existing approaches, our method does not save execution traces into any file. The `tracer` passes each method call of a trace sequentially to the `online specification miner`. The `online specification miner` learns class temporal specifications based on a probabilistic model. For each class, it

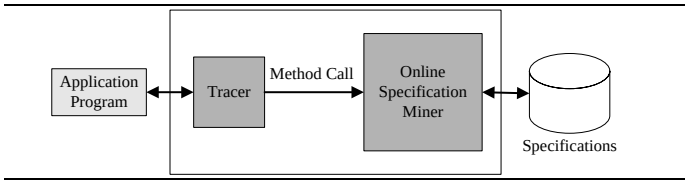

Figure 2. Working principle of our online specification mining technique.

first creates an empty specification described using the probabilistic model. Then, it evolves the probabilistic models persistently in terms of method calls passed by the `tracer`.

As we can see, our approach does not require loading all traces into memory. It refines existing probabilistic models based on a method call continuously. Therefore, compared with existing approaches, our method has lower space overhead. Furthermore, since method calls of any traces or application programs can be used to learn specifications, universal specifications may be achieved.

B. Collecting Program Execution Traces

To collect program execution traces, we should instrument application programs. Many approaches and frameworks exist to instrument Java applications statically or dynamically. We adopt Java agent technique, which is a service provided by Java since 1.5 [18]. Java agents can instrument classes at bytecode level. When a class is loaded, a Java agent catches the bytecode of this class on the fly. Then, it parses the class, injects new bytecodes. Finally, the instrumented class is returned back to the JVM.

To manipulate class bytecodes, we utilize a library `Javassist` [19], [20]. Compared with similar tools [21], [22], `Javassist` can provide the source level API, which enables programmers to edit a class file without knowledge of Java bytecodes. Furthermore, code can be inserted into class files in the form of Java source text and `Javassist` will compile it on the fly.

In order to collect program execution traces from an application program, we load a Java agent at startup using the `-javaagent` command-line switch. The agent will insert an `event writer` into the body of interested methods. Once the methods are called, the embedded `event writer` passes all necessary information regarding the method call to the specification miner for learning.

C. Mining Specification

1) Markov Chain with Final Probability

We mine class temporal specifications based on an extended Markov chain with final probability (MCF) [23]. MCF extends Markov chain by introducing a probability distribution over final states (final probability). The final probability is similar to initial probability. The difference is that final probability indicates which states a chance process should end with (rather than start from). The formal definition of MCF is given below.

DEFINITION 1 (Markov chain with final probability). A *Markov Chain with Final Probability* (*MCF*) *M* is a 4-tuple (Q, τ, π, γ), where Q is a set of states, $\tau : Q \times Q \to [0,1]$ is the *transition probability function*, which is always described using a *transition matrix P*, $\pi : Q \to [0,1]$ is the probability distribution over *initial states*. $\gamma : Q \to [0,1]$ is the probability distribution over *final states*. The functions π and γ must satisfy the requirements: $\forall q \in Q$, $\sum_{q \in Q} \pi(q) = 1$ and $\sum_{q \in Q} \gamma(q) = 1$.

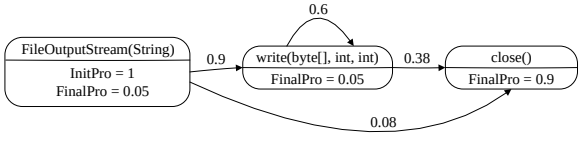

Figure 3. Class temporal specification of `FileOutputStream` described using MCF

As shown in the definition, MCF preserves most of characteristics of Markov chain, except violation of the requirement: $\forall q \in Q$, $\sum_{q' \in Q} \tau(q, q') = 1$, because of introduction of final states.

Relying on MCF, we can model class temporal specification by regarding states as methods and transitions as temporal relationships among methods. Consider the class temporal specification described using FSA illustrated in Figure 1, it can be described using a MCF as shown in Figure 3. The rounded rectangles are states labeled with method signatures above the line. Arrows denote transitions with transition probability labeled beside them. `InitPro` is the probability of a state to be initial state. `FinalPro` is the probability of a state to be final state. Actually, all the states have properties `InitPro` and `FinalPro`. We omit the ones whose value is zero. From the MCF, we can see that the usage of class `FileOutputStream` should start with a method call `FileOutputStream(String)`. At the end, methods `close()`, `FileOutputStream(String)` and `write(byte[],int,int)` may be called with a probability of 0.9, 0.05 and 0.05 respectively.

2) Online Specification Learning

Our approach learns class temporal specifications described using MCF in an online mode. It accepts a method call of an OUS as input and evolves existing specifications or creates a new one.

Let *R* be a repository of OUSs for learning, *M* be the MCF specification synthesized from *R*, *q* be a state of *M*, t_{ij} be a transition from state *i* to *j*. Our learning strategy represents *M* using a weighted directed graph G_M, where nodes and edges denote states and transitions respectively. In addition, the following properties are attached to G_M.

- *ouscount*(*M*) denotes the number of OUSs, which have been used to learn *M*.
- *emgcount*(*q*) denotes the total occurrence number of state (or method) *q* in *R*.
- *initcount*(*q*) denotes the count of *q* to be beginning method in all the OUSs of *R*.
- *finalcount*(*q*) denotes the count of *q* to be end method in all the OUSs of *R*.
- *emgcount*(t_{ij}) denotes the total occurrence number of method pair (*i*, *j*) in all the OUSs of *R*.

At the beginning, we initialize G_M to be an empty graph. Then, we pick up a method call from an OUS in *R* sequentially

and update G_M continuously until all OUSs have been processed. For each pair of method calls q and p received currently and previously, we update G_M based on the following strategy.

- if node q does not exist in G_M, add q to G_M or else update properties associates with q.
- if edge (p, q) does not exist in G_M, add (p, q) to G_M or else update properties associated with (p, q).
- if q is the end method of an OUS, update *ouscount*(M).

After that, we recompute probabilities τ , π and γ according to the following equations.

$$\tau(i, j) = emgcount(t_{ij}) / emgcount(i) \quad (1)$$

$$\pi(q) = initcount(q) / ouscount(M) \quad (2)$$

$$\gamma(q) = finalcount(q) / ouscount(M) \quad (3)$$

In words, $\tau(i, j)$ is the ratio between count of transition (i, j) and that of state i in all the OUSs used for learning. $\pi(q)$ is the ratio between number of OUSs beginning with state q and the total number of OUSs. $\gamma(q)$ is the ratio between number of OUSs ending with state q and the total number of OUSs.

3) Transformation from Probabilistic Model to Deterministic Model

MCF is a kind of probabilistic model, including frequent behaviors and infrequent behaviors. In order to use the mined specifications for program verification, we should prune away infrequent behaviors (noise) in the model and obtain a deterministic model. Chen et al. [23] proposed a deterministic model Class Interface Model (CIM) and showed that it is straightforward to transform MCF to CIM.

DEFINITION 2 (Class interface model). A Class Interface Model (CIM) M of class c is a 4-tuple (M, σ, S, F), where M is the set of public methods of c, $\sigma \subseteq M \times M$ is a binary relation on M, $S \subseteq M$ is the set of beginning methods, $F \subseteq M$ is the set of end methods. Let $p, q \in M$ be two methods, if they have the relation σ (denoted by $\sigma(p, q)$), it means that method p should be called preceding q.

A CIM of class c specifies that the usage of c should start from a method in S and then moves successively from a method m_i to m_j, where $\sigma(m_i, m_j)$, finally ends in a method of F. Any violations of the above rules are taken as errors.

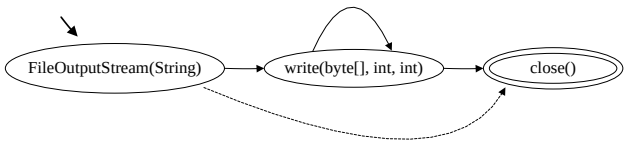

Figure 4. Class temporal specification of `FileOutputStream` described using CIM.

In order to transform MCF to CIM, we first prune away infrequent behaviors according to initial threshold (T_i), final threshold (T_f) and transition threshold (T_t), which are used to filter initial states, final states and transitions respectively. After that, we discard all the probabilities attached with states and transitions. In detail, given a MCF $\Omega : (Q, \tau, \pi, \gamma)$, we transform Ω to CIM $\Psi : (M, \sigma, S, F)$ in terms of the following rules:

- $\forall q \in Q$, add $\chi(q)$ to M, where $\chi: Q \rightarrow M$ is a function which maps a state in MCF to a method in CIM with method names the same as state labels.
- $\forall q \in Q$, if $\pi(q) \geq T_i$, add $\chi(q)$ to S.
- $\forall q \in Q$, if $\gamma(q) \geq T_f$, add $\chi(q)$ to F.
- $\forall i \in Q, j \in Q$, if $\tau(i, j) \geq T_t$, we have $\delta(i) = j$.

Figure 4 presents the CIM of class `FileOutputStream`, which is transformed from the MCF illustrated in Figure 3 based on threshold values T_i = 0.2, T_f = 0.2, T_t = 0.2. In the CIM, each ellipse represents a public method of the class. Arrows denote temporal relationships between pairs of methods. The methods with an arrow coming in from nowhere are beginning methods and those denoted graphically by a double ellipse are end methods. The dashed-line arrows represent the discarded transitions of MCF. As we can see, the previous MCF before transformation has three possible final states `FileOutputStream(String)`, `close()` and `write(byte[],int,int)` with a probability of 0.9, 0.05 and 0.05 respectively. The CIM discards the first and last final states because they are infrequent. In addition, the transition from state `FileOutputStream(String)` to `close()` is also pruned away due to a lower probability than T_t.

What should be noted is that results of transforming MCFs to CIMs largely depend upon values of thresholds. If thresholds are set too high, useful information will be discarded mistakenly. If thresholds are set too low, noise will remain. Even worse for our work, improper thresholds will cause unconnected CIMs. We employ the method proposed by Chen et al. [23] to compute threshold values, which can eliminate noise utmostly and obtain connected CIMs.

IV. EXPERIMENTS

In order to investigate the effectiveness of our technique, we implemented it in a prototype tool `ISpecMiner` and used the tool to mine specifications from several real-world applications. In this section, we first introduce subjects used in our experiments. Then, we present specifications mined by `ISpecMiner`.

A. Subjects

The subjects used in our experiment are listed in Table I, which consists of four real-world Java applications. We selected them based on the following criteria:

TABLE I. THE SET OF SUBJECTS

Subject	Version	Description	KLoC[a]	# Revisions	Create Date	Last Update Date
FreeMind	0.9	Mind-mapping software	22	6469	March, 2001	April, 2013
RapidMiner	5.3	Environment for machine learning and data mining	513	867	August, 2004	April, 2013
SQuirreL SQL Client	3.4	Java SQL client	253	3272	June, 2004	May, 2013
OpenProj	1.4	Project management software	120	1498	January, 2008	October, 2012

a. Kilo lines of code.

- Open source software. Though `ISpecMiner` is a dynamic specification miner and source code is not necessary, it is helpful for us to figure out problems encountered in the mining process and validate results.
- Mature software. Mature software contains fewer bugs than the unstable one. Thus, program execution traces with less noise can be collected, which is essential for dynamic mining tools to learn precise specifications. There exist many methods to measure the maturity of software. We perform the task based on a heuristic: if an application has been maintained for a long time and undergone a large number of revisions, we believe it is mature.
- Large-scaled software. Large-scaled software can provide abundant program execution traces for learning, which is the basis of mining useful program specifications.
- Applications coming from various domains. Applications from various areas can provide diverse program execution traces, which is a strong assurance for mined specifications to be complete.

B. Mining Specifications

In this experiment, we used `ISpecMiner` to mine specifications from the subject programs presented in Table I. We ran each application once with manual input data sequentially in the order of `FreeMind`, `RapidMiner`, `SQuirreL SQL Client` and `OpenProj`. After that, we examined the university of mined specifications achieved at the end of each run. The classes that we investigated are illustrated in Table 2. We selected these classes based on the following considerations: (1) they are widely used in various Java applications and well documented; (2) they are familiar to us; and (3) since their class temporal specifications have some distinguishing characteristics (such as the usage of a class should end with a method call `close()`), we can check their validity conveniently.

Figure 5 shows an example of mined specification for class `java.io.FileInputStream`, where (a), (b), (c) and (d) were achieved when we finished the run of subject programs `FreeMind`, `RapidMiner`, `SQuirreL SQL Client` and `OpenProj` respectively. As we can see, along with more applications used for mining, the specification grew universal, that is, more states and transitions were added to the specifications. For example, after the run of application

TABLE 2 INVESTIGATED CLASSES

	Class		Class
1	java.io.FileInputStream	6	java.io.InputStreamReader
2	java.io.BufferedReader	7	java.io.PushbackInputStream
3	java.io.FileOutputStream	8	java.io.FileReader
4	java.io.ByteArrayOutputStream	9	java.io.PrintWriter
5	java.io.BufferedWriter	10	java.util.Stack

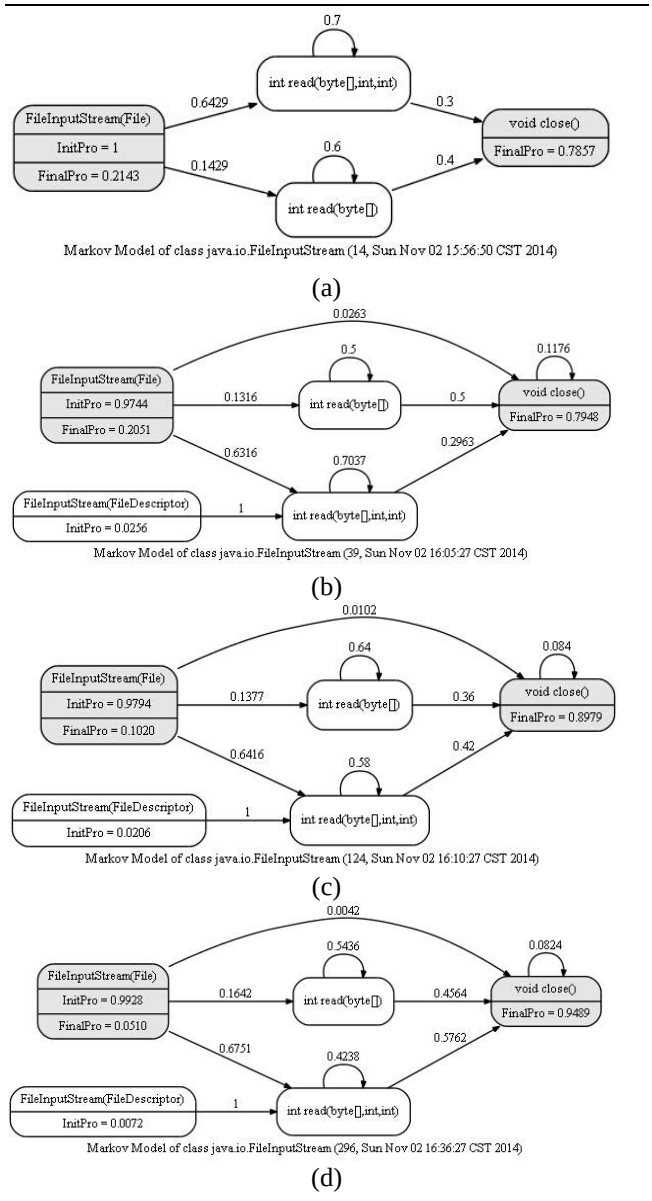

Figure 5. Example of mined probabilistic specification

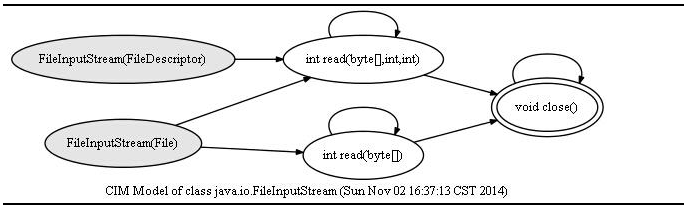

Figure 6. Example of mined deterministic specification

RapidMiner, a new state FileInputStream(FileDescriptor) and transition <FileInputStream(File), close()> shown in Figure 5 (b) were added to the previous specification illustrated in Figure 5 (a). Furthermore, since more applications were used to evolve the specification, probabilities of normal and abnormal behaviors (such as the FinalPro of state close() and that of state FileInputStream(File)) in the specification were increased and decreased respectively. Finally, the gap between probabilities of useful information and noise will become large, and then correct deterministic specifications can be achieved by transforming the final MCF to CIM. The specification of class FileInputStream described using CIM is illustrated in Figure 6, which is transformed from the final MCF under threshold values computed according to the method by [23]. After a close investigation, the CIM is correct and consistent with JDK documentations.

In conclusion, we used ISpecMiner to mine class temporal specifications from four real-world Java applications and examined specifications of 10 JDK classes. We found that our technique can refine mined specifications persistently. In addition, the probabilities of useful information will be enhanced, which is beneficial for transforming probabilistic models to correct deterministic models. ISpecMiner and other specifications mined in our experiment can be obtained at the URL http://ispecminer.com.

V. CONCLUSIONS

In this paper, we proposed an online program specification mining approach based on an extended Markov model. Different from existing approaches which work in a two-step mode, our method does not require saving collected program execution traces into a trace file. It first creates an empty probabilistic model for each class, and then evolves the probabilistic model persistently based on method calls in input traces. Since our approach does not require loading traces into memory at once, it has low space overhead. Additionally, if enough applications are provided for mining, universal specifications may be achieved.

ACKNOWLEDGMENT

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

REFERENCES

[1] A. Wasylkowski, A. Zeller, C. Lindig, Detecting object usage anomalies. Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM , Dubrovnik, 2007.

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

[3] M. Pradel and T.R. Gross, Automatic generation of object usage specifications from large method traces. Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, 2009.

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

[5] M. Pradel and T.R. Gross, Leveraging test generation and specification mining for automated bug detection without false positives. Proceedings of the 34th International Conference on Software Engineering, Zurich, Switzerland, 2012, 288-298.

[6] M.D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering, vol. 27, 2001, 99-123.

[7] M. Gabel and Z. Su, Javert: fully automatic mining of general temporal properties from dynamic traces. Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, Atlanta, 2008.

[8] J.H. Perkins and M.D. Ernst, Efficient incremental algorithms for dynamic detection of likely invariants. SIGSOFT Softw. Eng. Notes, vol. 29, 2004, 23-32.

[9] M.K. Ramanathan, A. Grama, and S. Jagannathan, Static specification inference using predicate mining. SIGPLAN Not., vol. 42, 2007, 123-134.

[10] S. Thummalapenta, and T. Xie, Alattin: mining alternative patterns for defect detection. Automated Software Engineering, vol. 18, pp. 293-323, 2011.

[11] D. Lo, G. Ramalingam, et al., Mining quantified temporal rules: formalism, algorithms, and evaluation. Science of Computer Programming, vol. 77, pp. 743-759, 2012.

[12] D. Chen, R. Huang, et al., Improving static analysis performance using rule-filtering technique. Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering, 2014.

[13] D. Lorenzoli, L. Mariani and M. Pezz, Automatic generation of software behavioral models. Proceedings of the 30th International Conference on Software Engineering, ACM, Leipzig, 2008.

[14] R. Alur, P. Cerny, et al., Synthesis of Interface Specifications for Java Classes. SIGPLAN Not., vol. 40, 2005, 98-109.

[15] J.E. Cook and A.L. Wolf, Discovering models of software processes from event-based data. ACM Trans. Softw. Eng. Methodol., 7(3), 1998, 215-249.

[16] G. Ammons, R. Bodik and J.R. Larus, Mining specifications. SIGPLAN Not., vol. 37, 2002, 4-16.

[17] M.D. Ernst, J.H. Perkins, et al., The Daikon system for dynamic detection of likely invariants. Science of Computer Programming, vol. 69, 2007, 35-45.

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

[19] S. Chiba and M. Nishizawa, An easy-to-use toolkit for efficient Java bytecode translators. Proceedings of the 2nd International Conference on Generative Programming and Component Engineering, Springer-Verlag, New York, 2003.

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

[21] ASM, http://asm.ow2.org.

[22] BCEL, http://commons.apache.org/proper/commons-bcel.

[23] D. Chen, R. Huang, et al., Ming class temporal specification dynamically based on extended Markov model. International Journal of Software Engineering and Knowledge Engineering, 2014, in press.