International Journal of Software Engineering
and Knowledge Engineering
Vol. 25, No. 3 (2015) 573–604
© World Scientific Publishing Company
DOI: 10.1142/S0218194015500047

World Scientific
www.worldscientific.com

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

Mining Class Temporal Specification Dynamically Based on Extended Markov Model

Deng Chen[*,†,¶], Rubing Huang[‡,‖], Binbin Qu[*,**], Sheng Jiang[*,††]
and Jianping Ju[§,‡‡]

[*]School of Computer Science and Technology
Huazhong University of Science and Technology
1037 Luoyu Road, Wuhan, P. R. China

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

[‡]School of Computer Science and Telecommunication Engineering
Jiangsu University, 301 Xuefu Road, Zhenjiang, P. R. China

[§]School of Mechanical Electronic and Information Engineering
Business College, Hubei University of Technology
634 Xiongchu Road, Wuhan, P. R. China
[¶]chendeng8899@hust.edu.cn
[‖]rbhuang@ujs.edu.cn
[**]bbqu@hust.edu.cn
[††]jwt@hust.edu.cn
[‡‡]gjdxjjp@126.com

Received 10 October 2013
Revised 24 February 2014
Accepted 25 April 2014

Class temporal specification is a kind of important program specifications especially for object-oriented programs, which specifies that interface methods of a class should be called in a particular sequence. Currently, most existing approaches mine this kind of specifications based on finite state automaton. Observed that finite state automaton is a kind of deterministic models with inability to tolerate noise. In this paper, we propose to mine class temporal specifications relying on a probabilistic model extending from Markov chain. To the best of our knowledge, this is the first work of learning specifications from object-oriented programs dynamically based on probabilistic models. Different from similar works, our technique does not require annotating programs. Additionally, it learns specifications in an online mode, which can refine existing models continuously. Above all, we talk about problems regarding noise and connectivity of mined models and a strategy of computing thresholds is proposed to resolve them. To investigate our technique's feasibility and effectiveness, we implemented our technique in a prototype tool ISpecMiner and used it to conduct several experiments. Results of the experiments show that our technique can deal with noise effectively and useful specifications can be learned. Furthermore, our method of computing thresholds provides a strong assurance for mined models to be connected.

Keywords: Program specification; class temporal specification; component interface; Markov model; specification mining; program verification.

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

1. Introduction

Class temporal specification plays an important role in program verification, evolution, understanding, etc. It is also referred to as component interface, object behavior model and object usage model [1]. A class temporal specification imposes temporal constraints regarding the order of calls of class interface 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. Client programs that violate such specifications do not obtain the desired behavior and may even crash the program [2]. However, class temporal specification is always implicit and undocumented. Even when available, there is no guarantee of their consistence, completeness, and correctness. Automatic specification mining [3–11] is a promising approach to resolve the problem.

Specification miner extracts common behaviors from client programs as specifications. It first collects method call sequences from applications statically or dynamically. Next, it splits method call sequences into a set of object usage scenarios (An *Object Usage Scenario* (*OUS*) is a method call sequence that contains calls to one object. For notational convenience, let C be a class, O is an object of C, we use $ous(O)$ to denote an OUS of O, where each element is a method called upon O). Finally, it reduces the problem of inferring specifications from the set of OUSs to the well known grammar inference problem [12] by regarding OUSs as sentences and specifications as languages. As a result, a specification is depicted using one or multiple finite state automata (FSA), where states represent states of involved objects and transitions represent method calls. Each path from an initial state to a final state forms a valid OUS. Figure 1 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 `FileOutputString (String)`. 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.

One of the disadvantages of the above method is that FSA is a kind of deterministic models with inability to tolerate noise. Noisy OUS is unavoidable, because

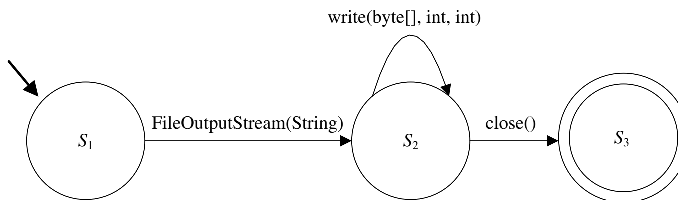


Fig. 1. FSA of class `java.io.FileOutputStream`. Each circle represents a state. Arrows denote transitions labeled with method signatures beside them. The state with an arrow coming in from nowhere is the initial state and that denoted graphically by a double circle is final state.

```
(1) FileInputStream fis = null;
(2) FileOutputStream fos = new FileOutputStream("filepath");
(3) byte[] buffer = new byte[1024];
(4) int count = 0;
(5) count = fis.read(buffer));
(6) fos.write(buffer, 0, count);
(5) count = fis.read(buffer));
(6) fos.write(buffer, 0, count);
(7) fis.close();
```

Fig. 2. Example of noisy OUSs caused by program bugs.

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

programs always contain bugs. Consider the Java program shown in Fig. 2, it contains two bugs. The first one is that object `fis` has not been correctly initialized at line 1. The second one is that `FileOutputStream fos` has not been closed after line 7. From the program, we can extract two OUSs: *ous*(`fis`): `<read, read, close>` and *ous*(fos): `<FileOutputStream, write, write>`, which have erroneous beginning and end method call respectively. These noisy OUSs with erroneous temporal relationships threaten accuracy of mined specifications.

To overcome the drawbacks in FSA, Ammons et al. [13] 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. As investigated, their approach is effective in tolerating noise. However, it has the following limitations: (1) since they employ an off-the-shelf PFSA learner, method call sequences should be transformed to scenario strings that can be accepted by the learner. It may be a nontrivial task for a long method call sequence; (2) their approach requires annotating programs to extract interaction scenarios, which reduces the practicality of the approach; and (3) results of transforming PFSA to NFA largely depend on computation of thresholds. Unconnected NFA may be generated with improper thresholds. For example, consider transforming the PFSA shown in Fig. 3(a) with a threshold 0.2, an unconnected NFA illustrated in Fig. 3(b) will be obtained. Obviously, the unconnected NFA is an invalid specification and useless for program verification and understanding. However, details of the topic are omitted in their work.

Markov chain is one of the traditional approaches to handle sequential data and is often used to model temporal patterns. It has a strong ability of tolerating noise. Successful applications have been demonstrated in domains of speech recognition [14, 15], signature verification [16–18] and DNA analysis [19]. Observed that, we propose to mine class temporal specifications dynamically based on a probabilistic model

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

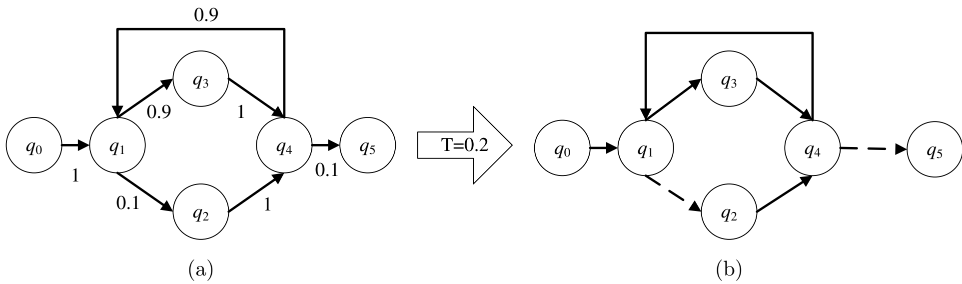


Fig. 3. Example of transforming PFSA to NFA with an improper threshold, which results in an unconnected model. (a) PFSA; (b) NFA transformed from the PFSA. Edge labels are omitted. The dashed-line arrows represent the discarded transitions.

extending from Markov chain. Compared with similar works, our technique has the following characteristics: (1) it learns program specifications in an online mode. Most existing dynamic program specification mining tools, such as `Daikon` [20] and `ADABU` [21], work in a two-step mode that a tracer component is first used to collect program execution traces in a data file and then an invariant detector or specification learner is employed to synthesize program specifications from the file. Problems in this approach are that specifications mined from different trace files will overwrite each other rather than integration and mined program specifications are biased to the trace file. We mitigate the problems by refining program specifications continuously using program execution traces collected from different applications; (2) rather than annotating programs, our technique can extract interaction scenarios directly in a more elegant way; (3) we transform probabilistic models into deterministic models based on computed thresholds, which can guarantee that the generated models are connected; and (4) although we focus on class temporal specifications in this work, our technique can be generalized to libraries and frameworks, where the sequence of calls to different classes matters the most.

To investigate our technique's feasibility and effectiveness, we implemented our technique in a prototype tool `ISpecMiner` and used it to conduct several experiments. Results of the experiments show that our technique can deal with noise effectively and mine useful specifications. Furthermore, the proposed method of computing thresholds is effective in obtaining connected models.

The contributions of this paper are:

- A probabilistic model extending from Markov chain is proposed to describe class temporal specifications.
- An algorithm is contrived to learn class temporal specifications in an online mode.
- A strategy of computing thresholds used to transform probabilistic models into deterministic models is introduced.
- A prototype tool `ISpecMiner` that implements our technique is presented.
- Several experiments are conducted using `ISpecMiner`.

The rest of this paper is organized as follows: Section 2 introduces the extended Markov model. Section 3 presents our online learning algorithm. Section 4 introduces the class interface model, which is used for program verification and manual inspection. Section 5 discusses our technique of transforming probabilistic models into deterministic models. Section 6 demonstrates results of our experiments. Sections 7 and 8 discuss the related work and present our conclusions.

2. Markov Chain with Final Probability

In this section, we introduce the Markov chain with final probability (MCF), which is utilized to model class temporal specifications in this work. We first present the definition of Markov chain and its application in modeling class temporal specifications, and then introduce the extended model with final probability, finally discuss advantages of our model.

2.1. *Markov chain*

Markov chain is a tool to study a specific type of chance process, in which, the outcome of a given experiment can affect the outcome of the next experiment. It is often used to handle sequential data and model temporal patterns. The formal definition of Markov chain is presented as follows.

Definition 1 (Markov Chain). A Markov chain M is a 3-tuple (Q, τ, π), where Q is a set of states, τ: $Q \times Q \rightarrow [0, 1]$ is the transition probability function, which is always described using a transition matrix P, π: $Q \rightarrow [0, 1]$ is the probability distribution over initial states. The functions τ and π must satisfy the requirements: $\forall\, q \in Q, \sum_{q' \in Q} \tau(q, q') = 1$ and $\sum_{q \in Q} \pi(q) = 1$.

In words, a Markov chain consists of a set of states $Q = \{q_1, q_2, \ldots, q_r\}$, $r \in N$. An initial state q_s $(1 \leq s \leq r)$ is designated with an initial probability $\pi(q_s)$. A chance process starts from q_s and moves successively from a state q_i to q_j $(1 \leq i \leq r$, $1 \leq j \leq r$, q_i and q_j may refer to a same state that is a state can move back to itself) with a transition probability $\tau(q_i, q_j)$.

It is a straightforward task to model class temporal specification using Markov chain by regarding states as methods and transitions as temporal relationships among methods. Take the specification of class `FileOutputStream` shown in Fig. 1 as an example, it can be described using a Markov chain illustrated in Fig. 4. From the model, we can see that a method call sequence (OUS) of class `FileOutputStream` must start from state `FileOutputStream(String)`, then traverses to state `write(byte[ ], int, int)` or `close()` with a probability of 0.9 and 0.1 respectively, finally ends in state `close()`.

2.2. *Markov chain with final probability*

Although Markov chain is able to model class temporal specifications, it is not appropriate for our purpose in some respects. One of the critical drawbacks is that we

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

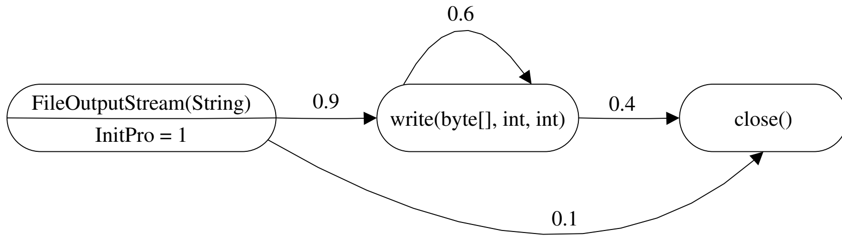


Fig. 4. Markov chain of class `FileOutputStream`. Rounded rectangles are states labeled with method signatures above the line. Arrows are transitions with transition probabilities labeled beside them. `InitPro` is the probability of a state to be initial states. Actually, all the states have property `InitPro`, we omit ones whose value is zero.

have no idea of which states an OUS should end in (we call this kind of state *final state*). Take the Markov chain shown in Fig. 4 as an example, we cannot tell whether state `FileOutputStream(String)` and `write(byte[ ], int, int)` are final states. However, final state is indispensable for us to detect many kinds of errors. For example, we can detect *resource leak error* according to whether the final method of an OUS of `FileOutputStream` is method `close()`. In order to resolve the problem, we attach a property to each state of a Markov chain to indicate probability of the state to be a final state. Formally, we give the definition of this kind of extended Markov chain as follows.

Definition 2 (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, τ: $Q \times Q \to [0, 1]$ 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, γ and τ must satisfy the requirements: $\forall\, q \in Q,\ \sum_{q \in Q} \pi(q) = 1,\ \sum_{q \in Q} \gamma(q) = 1$ and $\sum_{q' \in Q} \tau(q, q') \leq 1$.

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. Using MCF, class temporal specification can be modeled in a similar manner as that of Markov chain by regarding states as methods and transitions as temporal relationships among methods. Take the specification of `FileOutputStream` shown in Fig. 1 as an example, it can be described using a MCF illustrated in Fig. 5. From the MCF, we can see that an OUS of class `FileOutputStream` must start from state `FileOutputStream(String)` and may end in state `FileOutputStream(String)`, `write(byte[ ], int, int)` and `close()` with a probability of 0.01, 0.01 and 0.98 respectively.

Definition 3 (Connectivity of MCF). Given a MCF M: (Q, τ, π, γ), a state $q \in Q$ is called connected if q is included in a path of M from an initial state (a state q which satisfies $\pi(q) > 0$) to a final state (a state q which satisfies $\gamma(q) > 0$).

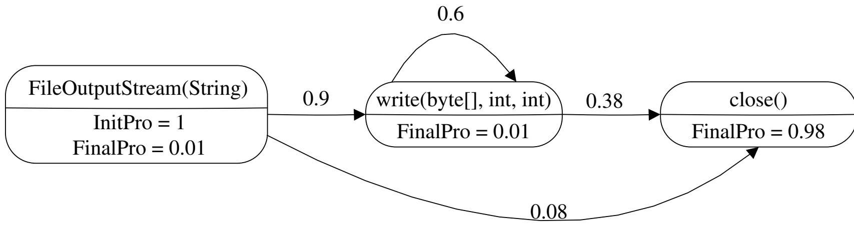


Fig. 5. Markov chain with final probability of class `FileOutputStream`. Rounded rectangles are states labeled with method signatures above the line. Arrows are transitions with transition probability labeled beside them. `InitPro` is the probability of a state to be initial states. `FinalPro` is the probability of a state to be final states. Actually, all the states should have properties of `InitPro` and `FinalPro`, we omit ones whose value is zero. It must be noted that, given a state q, the sum of its outgoing transition probabilities and final probability is different than one. As a matter of fact, it can be less than, equal to and greater than one.

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

Otherwise, it is called unconnected. A MCF is said to be connected if every state in the MCF is connected, or it is unconnected.

It must be noted that the connectivity of MCF is similar to that of graph. However, they are different in many aspects. Take the MCFs shown in (a) and (b) of Fig. 6 as an example, both of them are weak connected graphs but not connected MCFs. As we investigated, four different kinds of unconnected states may exist in MCFs, which are illustrated in Fig. 6.

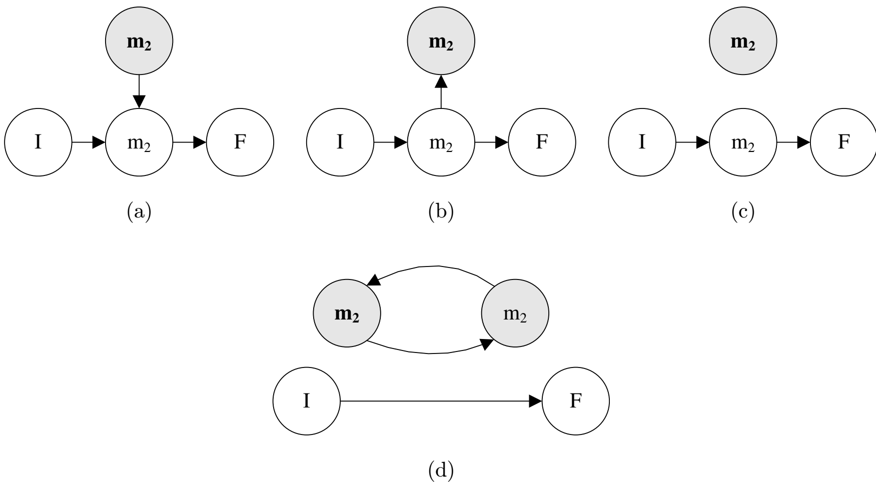


Fig. 6. Different types of unconnected states (filled with grey color) in MCF. (a) Unconnected state without incoming transitions, (b) unconnected state without outgoing transitions, (c) unconnected state without incoming nor outgoing transitions, (d) unconnected state with both incoming and outgoing transitions. I and F denote an initial and final state respectively and all the other states are neither initial nor final states. For simplicity, probabilities are omitted.

2.3. *Advantages of MCF*

Currently, existing techniques always use FSA and PFSA to model class temporal specification. Compared with those models, our model has the following advantages:

(1) Like any probabilistic models, MCF has inherent ability of tolerating noise. Generally, three kinds of noises may exist in OUSs, namely, *initial noise*, *sequential noise* and *final noise*. An OUS with initial and final noise has erroneous beginning and end method call respectively. For instance, *ous*(fis): <read, read, close> and *ous*(fos): <FileOutputStream, write, write> extracted from the buggy program shown in Fig. 2 contain initial and final noise respectively (the correct beginning and end method call should be FileInputStream() and close()). These two kinds of noises widely exist in mining specifications, especially when dynamic analysis approaches are employed, in which, specifications are synthesized from program execution traces. The problem is that incomplete traces with an incorrect end method call are always generated due to accidental interruptions of running applications. The sequential noise interferes with temporal properties between pairs of method calls. For example, we should first *write* data to a file, and then *close* FileOutputStream. If an OUS includes a method pair <close, write>, sequential noise occurs. MCF consists of three kinds of probabilities: transition probability (τ), initial probability (π) and final probability (γ), which can tolerate sequential, initial and final noise respectively.

(2) Since MCF extends from Markov chain, multiple applications can be exploited based on fundamentals of that. For instance, we can predict the most probable method following a method M after n steps based on the multi-step transition probability. In addition, given an OUS, we can find the most likely path in Markov chain relying on Viterbi algorithm [22]. Although, little benefit for the domain of specification mining has been found from these applications, it provides MCF the additional abilities that most of existing models cannot provide.

Just as Markov chain, MCF has different types, such as first-order and multiple-order MCF, discrete and continuous MCF, etc. In this paper, we concentrate on the first-order, discrete MCF and use it to model class temporal specifications.

3. MCF Online Learning Algorithm

Most existing dynamic specification mining tools work in two steps: (1) collecting program execution traces in a data file; (2) learning specifications from the trace file. This approach learns a model for each trace file. The problem is mined models may be biased to the input trace file. In this section, we present our online algorithm of learning MCFs from OUSs. Different from similar works, our algorithm does not save program execution traces in any file. It receives a method call of an OUS and then

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

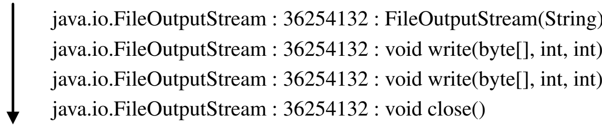


Fig. 7. Example of an OUS consisting of method events.

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

updates existing models or creates a new one. Since our algorithm evolves models continuously, more universal program specifications can be achieved.

Our learning algorithm is based on method events. A method event packages all the necessary information regarding a method call. Its format is given as follows:

$$ClassIdentifier : ObjectIdentifier : MethodIdentifier$$

where *ClassIdentifier*, *ObjectIdentifier* and *MethodIdentifier* are used to uniquely identify classes, objects and methods respectively. In this work, we utilize the full qualified name of classes, hash code computed according to memory address of objects and method signature as identifiers of classes, objects and methods respectively. Figure 7 illustrates an OUS consisting of four method events. As we can see, method events in an OUS have the same *ClassIdentifier* and *ObjectIdentifier*. Consequently, we can extract OUSs from method events by categorizing them according to *ClassIdentifier* and *ObjectIdentifier* conveniently. For notational convenience, let u be an OUS and e is a method event contained in u, we use $classid(e)$, $objid(e)$ and $methodid(e)$ to denote the *ClassIdentifier*, *ObjectIdentifier* and *MethodIdentifier* of e respectively. In addition, we call $objid(e)$ the identifier of u.

Let M be a MCF, q is a state, t_{ij} is a transition from state i to j, R is the repository of method events provided for learning. Our technique 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.

In order to learn MCF, our technique first instruments Java applications and collects method events. For each method event e, MCF learner distinguishes whether the corresponding MCF $classid(e)$ exists. If the MCF does not exist, an empty MCF will be created or the existing one is obtained. After that, we update the above properties regarding the MCF and recompute probabilities τ, π and γ. To update properties related to transitions, we record the previous method event of all the OUSs in a hash table H, where the key and value are the object identifier of OUSs

and previous method event respectively. In terms of the hash table, transitions can be obtained using $(H[objid(e)], e)$, where $H[k]$ returns the value in H with key k. The outline of our online learning algorithm is presented in Algorithm 1. From the algorithm, we can see that, $\tau(p, q)$ is the ratio between count of transition (p, q) and

Algorithm 1 MCF Online Learning Algorithm

Input: R_M: the repository of MCFs, H: hash table of previous method events, e: a method event.

Output: the updated repository of MCFs R_M.

Methords:

1) **if** R_M *not contains* MCF $classid(e)$ **then**
2) Add an empty model M to R_M.
3) **else**
4) Get existing MCF $classid(e)$ from R_M as M.
5) **end if**
6) **if** M *not contains* state $methodid(e)$ **then**
7) Add state q labeled with $methodid(e)$ to M.
8) **else**
9) Get state $methodid(e)$ from M as q.
10) **end if**
11) $emgcount(q) \leftarrow emgcount(q) + 1$.
12) **if** e is the beginning method call of an OUS **then**
13) $initcount(q) \leftarrow initcount(q) + 1$.
14) $ouscount(M) \leftarrow ouscount(M) + 1$.
15) **end if**
16) **if** e is the end method call of an OUS **then**
17) $finalcount(q) \leftarrow finalcount(q) + 1$.
18) **end if**
19) Get previous method event e' of OUS $objid(e)$ from H.
20) Get state $methodid(e')$ from M as p.
21) **if** M *not contains* transition (p, q) **then**
22) Add transition (p, q) to M.
23) **else**
24) Get transition (p, q) from M.
25) **end if**
26) Add entry $<objid(q), q>$ to H.
27) $emgcount(p, q) \leftarrow emgcount(p, q) + 1$.
28) $\tau(p, q) \leftarrow emgcount(p, q) \ / \ emgcount(p)$.
29) $\pi(q) \leftarrow initcount(q) \ / \ ouscount(M)$.
30) $\gamma(q) \leftarrow finalcount(q) \ / \ ouscount(M)$.

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

that of state p 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. As an example, we use the OUS shown in Fig. 7 for learning and achieve the following results (For simplicity, we use method names to denote methods and $p \to q$ denotes a transition from state p to q).

ouscount	1	*emgcount*(write → write)	1
emgcount(FileOutputStream)	1	*emgcount*(write → close)	1
initcount(FileOutputStream)	1	π (FileOutputStream)	1.0
finalcount(FileOutputStream)	0	π (write)	0
emgcount(write)	2	π (close)	0
initcount(write)	0	γ (FileOutputStream)	0
finalcount(write)	0	γ (write)	0
emgcount(close)	1	γ (close)	1.0
initcount(close)	0	τ (FileOutputStream → write)	1.0
finalcount(close)	1	τ (write → write)	0.5
emgcount(FileOutputStream → write)	1	τ (write → close)	0.5

It must be noted that all the properties can be extracted from OUSs directly and our algorithm is straightforward.

4. Class Interface Model

MCF is a kind of probabilistic model, including frequent behaviors and infrequent behaviors. In order to validate programs, we should prune away infrequent behaviors (noise) in the model. In this section, we introduce the class interface model, which is a deterministic model transformed from MCF.

A *Class Interface Model* (*CIM*) is like a FSA, which is composed of a set of states and transitions. Each state represents a method, and transition represents the temporal relationship between pairs of methods. The difference between CIM and FSA is that a CIM may have multiple initial states. As a matter of fact, we can transform a CIM to FSA by adding a common pseudo initial state and then connecting the state to all the initial states of the CIM. The formal definition of CIM is presented as follows.

Definition 4 (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 interface 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 as $\sigma(p, q)$), it means that method p should be called preceding q.

A CIM of class C specifies that each OUS of it must 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 8 presents a CIM of class `FileOutputStream`, which is transformed from the MCF illustrated in Fig. 5.

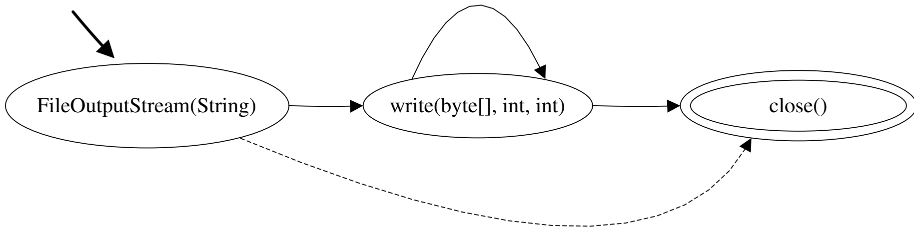


Fig. 8. CIM of class `FileOutputStream`. Each ellipse represents an interface method of class `FileOutputStream`. Arrows denote temporal relationships between pairs of methods. The methods with an arrow coming in from nowhere are beginning methods and those denoted graphically by a double ellipse are end methods. The dashed-line arrows represent the discarded transitions of MCF.

Definition 5 (Connectivity of CIM). Given a CIM C: (M, σ, S, F), a method $m \in M$ is called connected if m is included in a path of C from a beginning method to an end method. Otherwise, it is called unconnected. A CIM is said to be connected if every method in the CIM is connected, or it is unconnected.

5. Transforming MCF to CIM

In this section, we introduce our method of transforming MCF to CIM. We first provide an intuitive description of the technique and then discuss its main characteristics in detail.

5.1. *General approach*

In order to obtain CIM, we should prune away infrequent behaviors in MCF and discard probabilities attached with states and transitions. A general approach is filtering behaviors based on a threshold. What should be noted is that three kinds of behaviors need to be handled: initial states, final states and transitions. Therefore, we use three distinct probabilistic thresholds *initial threshold* (T_i), *final threshold* (T_f) and *transition threshold* (T_t) to filter them respectively (in the remainder of this paper, we call this group of thresholds *Threshold Vector* (TV), and denote it as (T_i, T_f, T_t)). Using the threshold vector, we transform a MCF $\Omega : (Q, \tau, \pi, \gamma)$ into CIM Ψ: (M, σ, S, F) according to the following rules, where the 4-tuple (Q, τ, π, γ) and (M, σ, S, F) are defined in Definitions 2 and 4 respectively.

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

More specifically, we first map each state in MCF to a method in CIM with method names the same as state labels. Next, we transform states with an initial

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

probability exceeding the threshold T_i and that with a final probability exceeding the threshold T_f into beginning and end methods respectively. Finally, we add transitions with a transition probability exceeding the threshold T_t to CIM.

Two special cases should be considered are as follows:

(1) $\exists\, q \in Q, \forall\, q' \in Q \wedge q' \neq q, 0 < \pi(q) < T_i, \tau(q', q) = 0$, that is, an initial state without incoming transitions and satisfying the condition $\pi(q) < T_i$.
(2) $\exists\, q \in Q, \forall\, q' \in Q \wedge q' \neq q, 0 < \gamma(q) < T_f, \tau(q, q') = 0$, that is, a final state without outgoing transitions and satisfying the condition $\gamma(q) < T_f$.

According to the above rules, these two kinds of states cannot be transformed into beginning and end methods respectively. However, this will cause unconnected CIMs similar to those shown in Figs. 6(a) and 6(b). To resolve the problem, we apply the following rules to deal with the above case (1) and (2) respectively.

(1) $\forall\, q \in Q, q' \in Q \wedge q' \neq q$, if $\tau(q', q) = 0$, add $\chi(q)$ to S.
(2) $\forall\, q \in Q, q' \in Q \wedge q' \neq q$, if $\tau(q, q') = 0$, add $\chi(q)$ to F.

The above rules state that, if a state in a MCF does not have incoming or outgoing transitions, then it must be a beginning or end method of transformed CIM despite its initial or final probability lower than the initial or final threshold. Our solution is based on the assumption that a method must be a beginning or end method if it occurs at the first or last position in all the OUSs including it.

As an example, we transform MCF illustrated in Fig. 5 into a CIM shown in Fig. 8 using threshold vector (0.2, 0.2, 0.2). As we can see, in the MCF before transformed, three possible final states exist: `FileOutputStream(String)`, `write(byte[ ], int, int)` and `close()` with a probability of 0.01, 0.01, and 0.98 respectively. The transformed CIM discards the former two 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.

5.2. *Computing initial and final thresholds*

It must be noted that results of transforming MCF to CIM largely depend upon threshold vector. As a matter of fact, it is a common concern of all approaches based on statistical analysis. 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 threshold vector will cause unconnected CIM. In the remainder of this section, we discuss the problem and propose an algorithm to compute thresholds.

Different from many statistical approaches that set thresholds empirically to fixed values, our probabilistic thresholds are computed from MCFs that different MCFs have their own thresholds. Assume q to be a common state contained in a set of MCFs $\{\mathrm{MCF}_1, \mathrm{MCF}_2, \ldots, \mathrm{MCF}_n\}$, where MCF_n denotes a MCF with n states. It is obvious that $\pi(q)$ of MCF_n is most probably smaller than that of MCF_1, when $n \gg 1$.

However, we cannot make the conclusion that q is an initial state in MCF_1 but not in MCF_n. The problem here is that we should not filter the initial states according to a same criterion for different MCFs and similar conclusions can be achieved as to final states and transitions.

We compute initial and final thresholds based on a heuristic that the probability of an infrequent behavior must be lower than the average probability. Given MCF_n, q is a state of it, the mean of initial and final probability of q is $1/n$. Inspired by the heuristic, we introduce two threshold factors TF_i and TF_f which satisfy the requirements: $TF_i \leq 1$, $TF_f \leq 1$ and compute T_i and T_f based on the following equations:

$$T_i = TF_i/n \tag{1}$$

$$T_f = TF_f/n \tag{2}$$

It must be noted that the maximum values of T_i and T_f are the means of initial and final probability respectively, which are achieved when the threshold factors TF_i and TF_f are set to 1.

5.3. *Computing transition threshold*

The computation of transition threshold is more complicated than that of initial and final thresholds, because an improper T_t will cause the transformed CIM unconnected. In this subsection, we first discuss the connectivity problem in transformation and then introduce our algorithm of computing transition threshold in detail.

5.3.1. *Connectivity problem in transformation*

Lemma 1. *Assume M to be a MCF learned from a set of OUSs, M must be connected.*

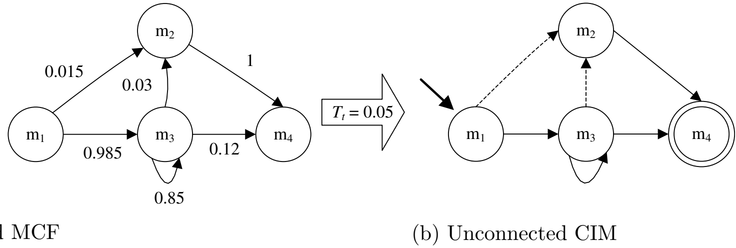
Proof (Proof by Contradiction). Assume that M: (Q, τ, π, γ) is an unconnected MCF learned from a set of OUSs R. $q \in Q$ is an unconnected state which is labeled with a method signature l. Since q is unconnected, for each $s \in R$, if s includes l, then s must be an OUS without beginning or end method, because every beginning and end method corresponds to an initial and final state in MCFs respectively. However, as we all know, an OUS is linear with both beginning and end method, which is a contradiction. □

Although the mined MCFs are proved to be connected, the connectivity property may change when transformed to CIMs. Take the connected MCF illustrated in Fig. 9(a) as an example, when carrying out transformation with a transition probability of 0.05, we obtain an unconnected CIM shown in Fig. 9(b). This is extremely a bad case that we should avoid, because an unconnected CIM is incorrect and useless. The reason for this is that useful transitions are discarded because of employment of a transition threshold higher than what it should be, or in other words, there exists an underlying maximum transition threshold and T_t should be lower than that.

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

Probabilities of States:

State	InitPro	FinalPro
m_1	1	0
m_2	0	0
m_3	0	0
m_4	0	1



(a) Connected MCF (b) Unconnected CIM

Fig. 9. Unconnected CIM will be obtained in transformation if an improper T_t is used. (a) MCF which is required to be transformed. (b) Unconnected CIM which is generated by transforming the MCF using a transition probability of 0.05. The transformation is performed with parameters $TF_i = TF_f = 0.1$. The dashed lines represent the discarded transitions of MCF.

Definition 6 (Maximum Connected Transition Threshold). Assume M to be a connected MCF, C is the CIM transformed from M based on a transition threshold T_t, the Maximum Connected Transition Threshold (MCTT) of M denoted as MCTT (M) is an extreme of T_t, such that if $T_t \leq MCTT(M)$, C is connected. Otherwise C is unconnected.

5.3.2. *Maximum-threshold connected spanning sub-PMCF*

From Sec. 5.3.1, we realized that we can resolve the connectivity problem in transformation using a transition threshold lower than the maximum connected transition threshold. In this subsection, we introduce the concept of *Maximum-threshold Connected Spanning Sub-PMCF* and demonstrate that the problem of computing maximum connected transition threshold can be reduced to that of constructing maximum-threshold connected spanning sub-PMCF.

Definition 7 (Pseudo MCF). A Pseudo MCF (PMCF) M is a 4-tuple (Q, τ, π, γ) similar to MCF. The difference is that PMCF may violate requirements of MCF: $\forall q \in Q, \sum_{q \in Q} \pi(q) = 1$ and $\sum_{q \in Q} \gamma(q) = 1$. Therefore, MCF is a special type of PMCF. In addition, the connectivity definition of MCF is also appropriate for PMCF.

Property 1. *Given a connected PMCF P:* (Q, τ, π, γ), we have:

- $\forall q \in Q \wedge \pi(q) = 0, \exists q' \in Q,$ *such that* $\tau(q', q) \neq 0$
- $\forall q \in Q \wedge \gamma(q) = 0, \exists q' \in Q,$ *such that* $\tau(q, q') \neq 0$

Proof. The property can be proved straightforwardly based on Definition 3 and we omit the details. □

Definition 8 (Sub-PMCF). Given a PMCF P: (Q, τ, π, γ), PMCF P': $(Q', \tau', \pi', \gamma')$ is said to be a Sub-PMCF of P, if it satisfies the following requirements:

- $\forall q \in Q', \pi'(q) = \pi(q) \vee \pi'(q) = 0$

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

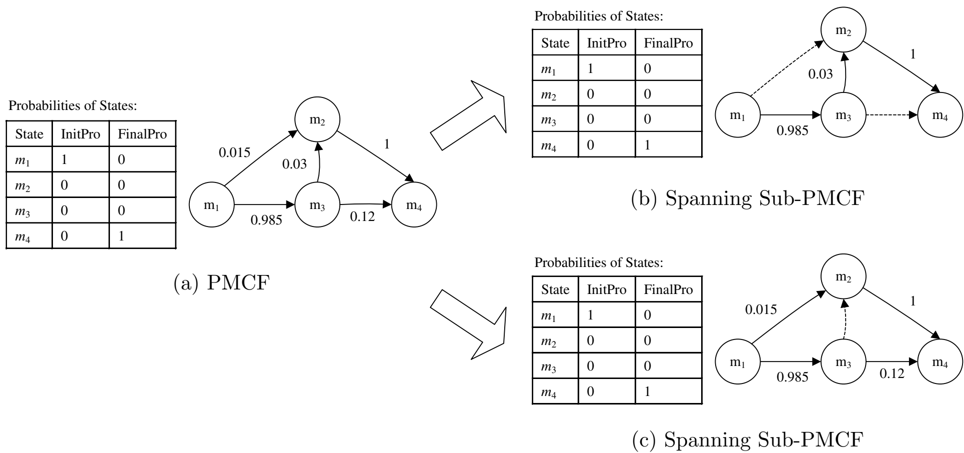


Fig. 10. Spanning Sub-PMCF. (a) PMCF. (b) and (c) are two spanning sub-PMCFs of (a). The dashed lines represent the discarded transitions of PMCF.

- $\forall\, q \in Q', \gamma'(q) = \gamma(q) \vee \gamma'(q) = 0$
- $\forall i \in Q', j \in Q', \tau'(i,j) = \tau(i,j) \vee \tau'(i,j) = 0$

Additionally, we call P' a Spanning Sub-PMCF of P, if $Q = Q'$.

As an example, Fig. 10 presents two spanning sub-PMCF in (b) and (c) of the PMCF in (a).

Definition 9 (Maximum-Threshold Connected Spanning Sub-PMCF). Given a connected PMCF P: (Q, τ, π, γ), the Minimum Transition Probability (MINTP) of P is defined as follows: $\forall\, i \in Q, j \in Q, MINTP(P) = \min(\tau(i,j))$. Let P_m: $(Q, \tau_m, \pi_m, \gamma_m)$ be a connected spanning sub-PMCF of P. P_m is said to be the Maximum-Threshold Connected Spanning Sub-PMCF of P, if for any connected spanning sub-PMCF P_i: $(Q, \tau_i, \pi_i, \gamma_i)$ of P, we have $MINTP(P_m) \geq MINTP(P_i)$. In addition, we call $MINTP(P_m)$ the Maximum Spanning Transition Probability (MAXSTP) of P which is denoted as MAXSTP(P).

Theorem 1 (MAXSTP is the MCTT). *Given a connected MCF M*: (Q, τ, π, γ), *we have* $MCTT(M) = MAXSTP(M)$.

Proof. Proof is straightforward based on Definitions 6 and 9. We omit the details. □

5.3.3. *Judgement of connected PMCF*

From Theorem 1, to obtain MCTT of a MCF, we should first find the maximum-threshold connected spanning sub-PMCF of it. In this part, we present several lemmas which are the basis of our solution to the problem.

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

Lemma 2. *Given a PMCF P*: (Q, τ, π, γ), *if P satisfies the following requirements*, *P must be connected.*

- $\forall\, q \in Q \wedge \pi(q) = 0$, q has at least one incoming transition
- $\forall\, q \in Q \wedge \gamma(q) = 0$, q has at least one outgoing transition
- P contains no cycle

Proof (Proof by Contradiction). Assume P: (Q, τ, π, γ) to be an unconnected PMCF, which satisfies the above requirements, $q \in Q$ is an unconnected state. Two cases should be considered for a state to be unconnected.

(1) (q is unconnected with initial states that is there does not exist a path from an initial state to q) According to the above requirements, there must exist a transition $t_{q_1 q}$, which starts from a state q_1 and ends in q. Since q is unconnected with initial states, q_1 must be unreachable from initial states. We repeat the above process by regarding q_1 as q. Finally, a path H: $\cdots q_n \rightarrow q_{n-1} \cdots q_2 \rightarrow q_1 \rightarrow q$ with infinite length will be obtained. However, P has a finite set of states, thus H must contain cycles, which is a contradiction.

(2) (q is unconnected with final states, that is, there does not exist a path from q to a final state) Proof is similar to that of the first case. We omit the details.

It should be noted that q may be of different types: (1) initial state, (2) final state and (3) other types. Whatever type it is, the reason for it to be unconnected must fall into the above categories. For example, if q is an unconnected final state, it must be unconnected with initial states (first case above). If q is an unconnected initial state, it must be unconnected with final states (second case above). □

Definition 10 (Benign Cycle). Given a PMCF P: (Q, τ, π, γ), C is a cycle in P, we call C a Benign Cycle, if C has at least one incoming and outgoing transition from or to a external state of C (that is, there exist two transitions t_{ij} and t_{pq}, where $i \notin C, j \in C, p \in C, q \notin C$). We call a cycle Malign Cycle, if it is not a benign cycle.

Lemma 3. *Given a PMCF P*: (Q, τ, π, γ), *if P satisfies the following requirements*, *P must be connected.*

- $\forall\, q \in Q \wedge \pi(q) = 0$, *$q$ has at least one incoming transition*
- $\forall\, q \in Q \wedge \gamma(q) = 0$, *$q$ has at least one outgoing transition*
- *All cycles in P are benign cycles*

Proof. Contract each cycle in P into a pseudo-state and form a new PMCF P': $(Q', \tau', \pi', \gamma')$. Because all cycles in P are benign cycles, the pseudo-states must have at least one incoming and outgoing transition. According to Lemma 2, P' must be connected. Therefore, $\forall\, q \in Q'$, q is connected, or in other words, $\forall\, q \in Q$, if q is not inside any cycles, q must be connected. Consequently, to prove the connectivity of P, what we need to do is demonstrate that all states inside cycles are connected. Let C be a benign cycle in P, according to Definition 10, there must exist two transitions

(q_1, q_{c1}) and (q_{c2}, q_2), where $q_1 \notin C$, $q_{c1} \in C$, $q_{c2} \in C$, $q_2 \notin C$. For each state q in C, q must be included in a path from q_{c1} to q_{c2}, because they are in a same cycle. Combined with the transitions $(q_1,\ q_{c1})$ and $(q_{c2},\ q_2)$, we obtain a path $q_1 \to q_{c1} \to \cdots q \cdots \to q_{c2} \to q_2$. On the other hand, since q_1 and q_2 are external states of C, they must be connected. Based on the above analysis, q is connected. Since all states in P are connected, we have P is connected. □

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

5.3.4. *Computing MAXSTP*

Based on above analysis, we propose a *Selection of Maximum Transition* (*SMT*) algorithm to construct the maximum-threshold connected spanning sub-PMCF. For notational convenience, we first introduce several concepts. Given a PMCF P: (Q, τ, π, γ), $q \in Q$ is a state of P, the *Maximum Incoming Transition Probability* (*MAXITP*) and *Maximum Outgoing Transition Probability* (*MAXOTP*) of q are defined as follows:

$$MAXITP(q) = \max_{q' \in Q}(\tau(q', q)) \tag{3}$$

$$MAXOTP(q) = \max_{q' \in Q}(\tau(q, q')) \tag{4}$$

The transition with MAXITP and MAXOTP are called *Maximum Incoming Transition* (*MAXIT*) and *Maximum Outgoing Transition* (*MAXOT*) respectively.

Assume P to be connected, the maximum-threshold connected spanning sub-PMCF of P can be constructed using the SMT algorithm, which works as follows:

(1) Discard all self-transitions (which start from and end in a same state) in P.
(2) $\forall\, sq \in Q$, add $MAXIT(q)$ to a set of transitions Γ.
(3) $\forall\, q \in Q$, if q does not have an outgoing transition in Γ, add $MAXOT(q)$ to Γ.
(4) If no malign cycles formed in Γ, go to step 5. Otherwise, for each malign cycle C in Γ, we perform the following tasks: (1) if C does not have incoming transitions, add transition t_{in} to Γ; (2) if C does not have outgoing transitions, add transition t_{out} to Γ, where t_{in} and t_{out} satisfy the following equations: $\tau(t_{in}) = \max_{q' \notin C \wedge q \in C}(\tau(q', q))$, $\tau(t_{out}) = \max_{q \in C \wedge q' \notin C}(\tau(q, q'))$.
(5) PMCF P_m: (Q, τ_m, π, γ) is the maximum-threshold connected spanning sub-PMCF of P, where $\forall\, i \in Q, j \in Q$, τ_m is defined as follows:

$$\tau_m(i,j) = \begin{cases} \tau(i,j) & t_{ij} \in \Gamma \\ 0 & t_{ij} \notin \Gamma \end{cases} \tag{5}$$

Theorem 2 (Validity of SMT Algorithm). *The PMCF produced by SMT algorithm is the maximum-threshold connected spanning sub-PMCF.*

Proof. Given a PMCF P: (Q, τ, π, γ), apply SMT algorithm to P and obtain a PMCF P_m: $(Q_m, \tau_m, \pi_m, \gamma_m)$. According to the working principle of SMT algorithm,

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

P_m must be a spanning sub-PMCF of P, because $Q = Q_m$ and transitions of P_m is a subset of P. In order to prove the validity of SMT algorithm, we should demonstrate the connectivity and maximum properties of P_m respectively.

(1) (Connectivity property) From the steps of SMT algorithm, P_m is a PMCF without malign cycles and all states in P_m apart from initial states have at least one incoming transition and that except final states have at least one outgoing transition. According to Lemma 3, P_m must be connected.
(2) (Maximum property, proof by contradiction) Assume that PMCF $P_{m'}$: $(Q_{m'}, \tau_{m'}, \pi_{m'}, \gamma_{m'})$ is a connected spanning sub-PMCF of P and $MINTP(P_{m'}) > MINTP(P_m)$. Let $MINTP(P_{m'}) = \tau_{m'}(q'_1, q'_2)$, $MINTP(P_m) = \tau_m(q_1, q_2)$, according to the working principle of SMT algorithm, transition (q_1, q_2) is either the MAXIT of q_2 or the MAXOT of q_1. Assume that $MAXITP(q_2) = \tau_m(q_1, q_2)$ in P, because $P_{m'}$ is connected, in terms of Property 1, there must exist a transition (q', q_2) in $P_{m'}$ and $\tau_m(q_1, q_2) \geq \tau_{m'}(q', q_2)$. Since $MINTP(P_{m'}) > MINTP(P_m)$, we have $MINTP(P_{m'}) > \tau_{m'}(q', q_2)$, which is a contradiction. A similar contradiction will be achieved in the case of $MAXOTP(q_1) = \tau_m(q_1, q_2)$. □

Our strategy of computing transition threshold can maintain connectivity of transformed CIMs. However, it cannot deal with sequential noise. To resolve the problem, we apply the heuristic used to deal with initial and final noise to sequential noise, that is, the probability of an infrequent transition must be lower than the average transition probability. Give a MCF M with n states, the average transition probability is $1/n$ (which is obtained by assuming that a state has outgoing transitions with all the states). Based on the above analysis, we compute T_t of M as follows:

$$T_t(M) = \min(MCTT(M), 1/n) \times TF_t \tag{6}$$

where TF_t is the threshold factor of transition probability, which should be assigned a value between 0 and 1 (we call the threshold factor of transition probability combined with that of initial and final probability *Threshold Factor Vector* (TFV), and denote it as (TF_i, TF_f, TF_t)). As we can see, transition threshold computed according to the above equation can maintain connectivity of transformed CIMs as well as deal with sequential noise.

5.3.5. *Example of transformation using computed transition threshold*

Take the MCF M shown in Fig. 11(a) as an example, we apply SMT algorithm to it step by step. After the third step, the transition set Γ contains the following elements:

Start State	End State	Transition Probability
m_3	m_2	0.03
m_1	m_3	0.985
m_2	m_4	1

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

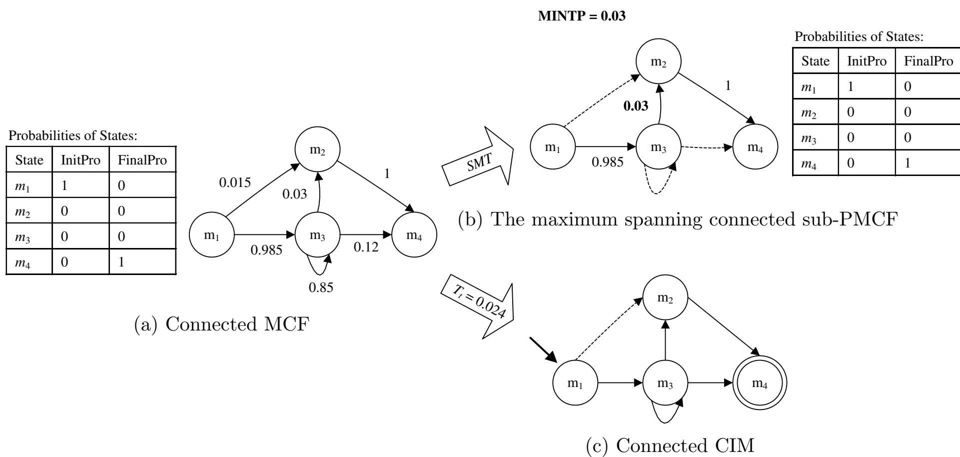


Fig. 11. Transforming MCF to connected CIM using computed transition threshold. (a) MCF required to be transformed. (b) The maximum-threshold connected spanning sub-PMCF which is generated using SMT algorithm. (c) Connected CIM transformed from the MCF shown in (a). The transformation is performed with a TFV: (0.8, 0.8, 0.8). The dashed lines represent discarded transitions of MCF.

Since no cycle is formed in Γ, we obtain the maximum-threshold connected spanning sub-PMCF P of M, which is presented in Fig. 11(b). In addition, we have $MCTT(M) = MINTP(P) = 0.03$. On the other hand, the average transition probability of M is $1/4 = 0.25$. Suppose we transform the MCF to CIM using a TFV: (0.8, 0.8, 0.8), according to Eqs. (1), (2) and (6), we have TV: (0.2, 0.2, 0.024). The CIM transformed using the computed TV is shown in (c), which is connected.

6. Experiments

To evaluate our technique, we implemented our technique in a prototype tool `ISpecMiner` and used it to conduct experiments. In this section, we first introduce `ISpecMiner`. Next, we present subjects that we used in evaluation. Then, we conduct several experiments and investigate the following issues.

- How to choose values of threshold factors (TF_i, TF_f, TF_t)?
- How effective of our approach to deal with noise?
- Whether our approach can guarantee connectivity of generated CIMs?
- Whether our method can mine useful class temporal specifications?

Finally, we discuss limitations of our approach.

6.1. *Prototype tool ISpecMiner*

`ISpecMiner` is a dynamic program specification mining tool developed based on `Java 1.6`. It instruments bytecodes of Java classes at load-time leveraging `Java`

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

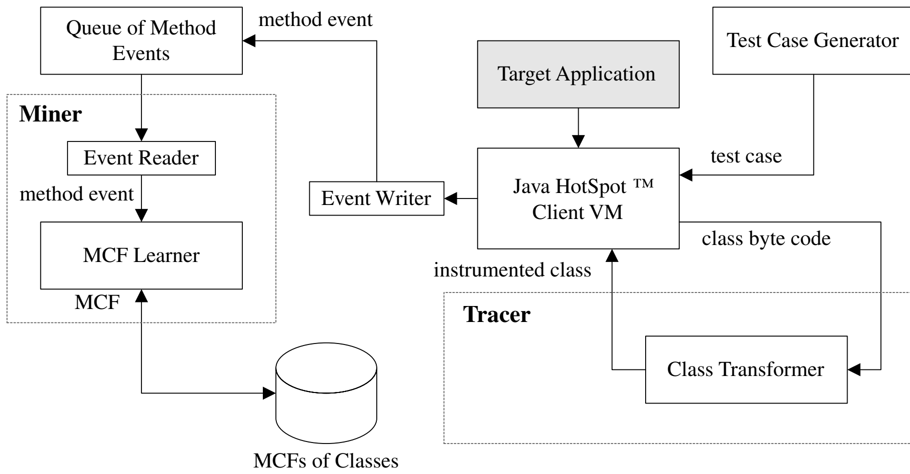


Fig. 12. Composition of `ISpecMiner`.

`agent` technique and then learns MCF models from program execution traces generated by running the instrumented applications.

The composition of `ISpecMiner` is shown in Fig. 12. It consists of two main components: `tracer` and `miner`. The `tracer` is a Java agent which receives class bytecodes from JVM and then returns the instrumented bytecodes to JVM for running. The code that `tracer` weaves into each method is an `event writer`. Once an instrumented method is called, `event writer` will push a `method event` into a queue. The `queue of method events` is a cache shared by the `tracer` and `miner` (the `tracer` produces events and `miner` consumes events). Whenever it is nonempty, the `miner` picks up a `method event` from the queue through the `event reader`, and then passes it to the `MCF learner` for learning. If the `method event` belongs to an existing MCF, the learner updates it or a new one is created. As we can see, this architecture is very flexible. Especially that, the communication protocol between the `tracer` and `miner` can be replaced freely by providing a new pair of `event writer` and `reader`.

Instrumentation technique is crucial for dynamic program specification mining tools. Many approaches and frameworks exist to instrument Java applications statically or dynamically. We adopt `Java agent` technique combined with `Javassist` to instrument Java classes at the bytecode level dynamically. `Java agent` technique is a service provided in the JDK package of `java.lang.instrument` which allows a nearly complete control over classes of any given application. Observed that `Java agent` does not provide facilities to manipulate class bytecodes, we employ a bytecode manipulation tool `Javassist` [23–26]. Compared with similar tools [27, 28], `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 use `ISpecMiner` to mine specifications from a Java application, we should use the `-javaagent` command-line switch to start the application. Details are presented as follows:

$$\text{java} - \text{javaagent} : \textit{agentpath} = \textit{mainclass apppath}$$

where *agentpath* is the path of `ISpecMiner`, *apppath* is the path of target application,*mainclass* should be designated with name of the main class in target application. `ISpecMiner` is able to instrument most of Java classes (except for several JDK classes due to some technical reasons), whose names are specified using regular expressions in configuration. For each class, `ISpecMiner` generates two kinds of models: MCF and CIM. MCFs are evolved continuously and CIMs are transformed from the latest MCFs when necessary. `ISpecMiner` can be obtained at the URL http://www.ispecminer.com.

6.2. *Subjects*

Applications that we used in our experiments are listed in Table 1. All of them are Java applications. We select them 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 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.
- Applications of a large size. Large-scale software can provide abundant OUSs for learning, which is the basis of mining useful program specifications.
- Applications coming from various domains. Applications from various areas may cover every usage detail of target classes, which is a strong assurance for mined specifications to be comprehensive and accurate.

6.3. *Experiment 1: Transforming MCF to CIM with different values of TFV*

In this work, we proposed a method to deal with noise by assuming that probability of noise must be lower than the average probability. Theoretically, our method can protect useful information from being discarded mistakenly. However, low values of TFV will still result in noise. In this experiment, we used `ISpecMiner` to mine MCFs from several real-world applications and then transform them to CIMs with a group of different values of TFV. Our purpose is to investigate how effective of our

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

Table 1. Applications used in experiments. # *Revisions*: number of revisions; *Create Date*: the date that applications were created; *Last Update Date*: the date that applications were last updated. We use # *Revisions*, *Create Date* and *Last Update Date* to approximately measure the maturity of software, that is, we assume software to be mature if it has been maintained for a long time and undergone a large number of revisions.

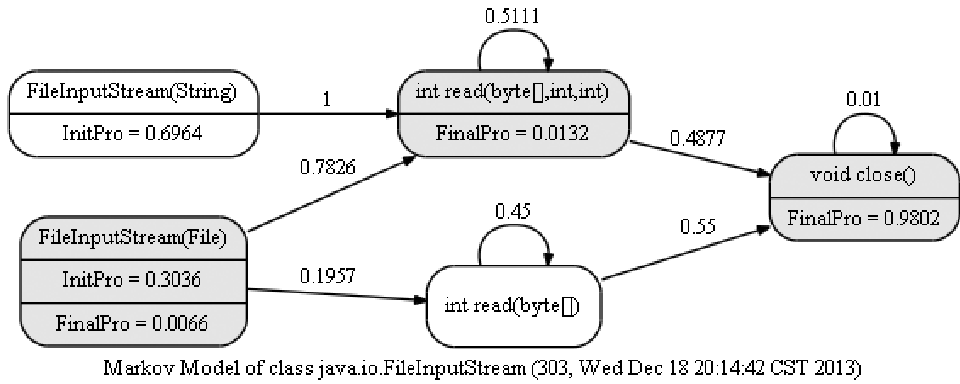
Subject	Version	Description	# Revisions	Create date	Last update date
FreeMind	0.9	Mind-mapping software	6469	March, 2001	April, 2013
RapidMiner	5.3	Environment for machine learning and data mining	867	August, 2004	April, 2013
SQuirreL SQL Client	3.4	Java SQL client	3272	June, 2004	May, 2013
OpenProj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

approach to deal with noise and how to choose proper values of TFV, which can eliminate noise utmostly.

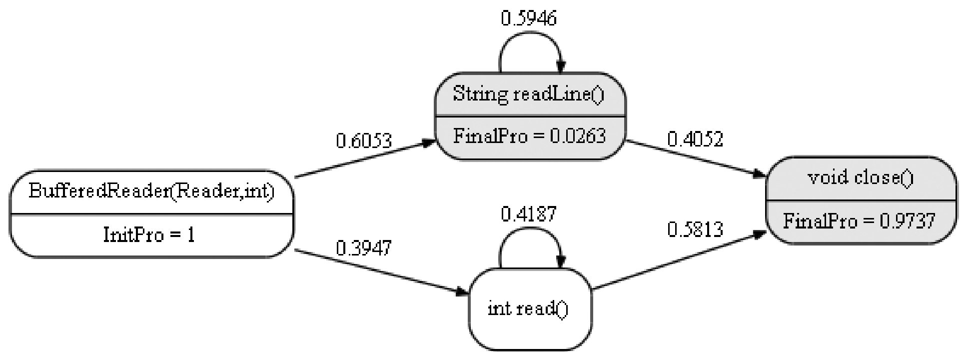
Our experiment first learns MCFs from real-world applications using `ISpecMiner`. The subject applications that we used for mining are listed in Table 1. To collect enough OUSs and avoid bias, we run each application five times with different manual input data. `ISpecMiner` was configured to mine MCFs of 10 JDK classes. The classes and number of OUSs collected for learning the MCFs are presented in Table 2. We selected these classes based on the following considerations: (1) classes that are used widely in various Java applications and well documented; (2) classes that are familiar to us; (3) classes with distinguishing characteristics, such as the usage of a class should end in a method call `close()`. Figure 13 illustrates several of the mined MCFs (MCFs of all the classes, please refer to the website http://www.ispecminer.com). After that, we transformed mined MCFs to CIMs using a set of TFV spanning from 0.1 to 1.0 with a step of 0.1. For each TFV, we investigated the average *goodness* of all the transformed CIMs. According to the distribution of average goodness, an interval of TFV is obtained, which can lead to perfect CIMs. We measured the goodness of a CIM using the sum of false positives and false negatives. A *false positive* is a correct behavior (initial state, final state or transition), which is discarded mistakenly in transformation. A *false negative* is an erroneous behavior (noise) contained in transformed CIMs (which should be pruned away). A

Table 2. Mined MCFs of JDK classes. # OUSs: number of OUSs collected for learning the MCF of a class.

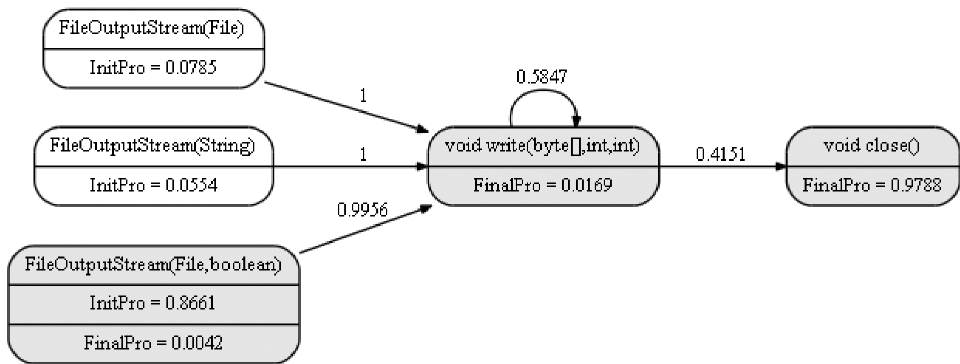
Class	#OUSs	Class	#OUSs
`java.io.FileInputStream`	303	`java.io.FileOutputStream`	236
`java.io.FileReader`	213	`java.io.PushbackInputStream`	30
`java.io.InputStreamReader`	291	`java.io.BufferedReader`	38
`java.io.ByteArrayOutputStream`	19	`java.io.PrintWriter`	174
`java.io.BufferedWriter`	27	`java.util.Stack`	268



(a)



(b)

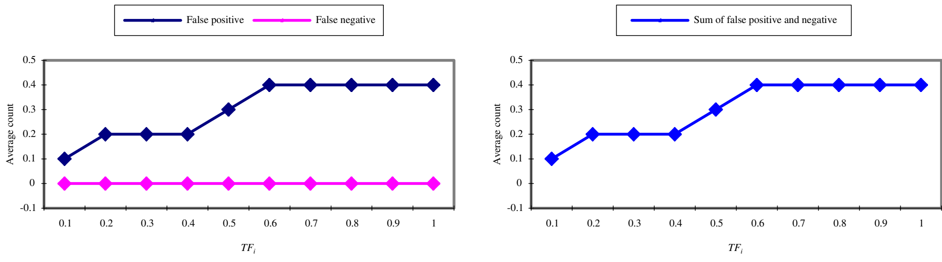


(c)

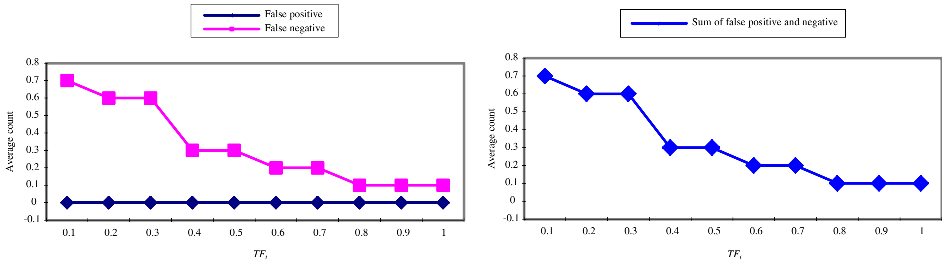
Fig. 13. Several of the mined MCFs. (a): MCF of class `java.io.FileInputStream`; (b): MCF of class `java.io.BufferedReader`; (c) MCF of class `java.io.FileOutputStream`. Rounded rectangles filled with gray color are final states.

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

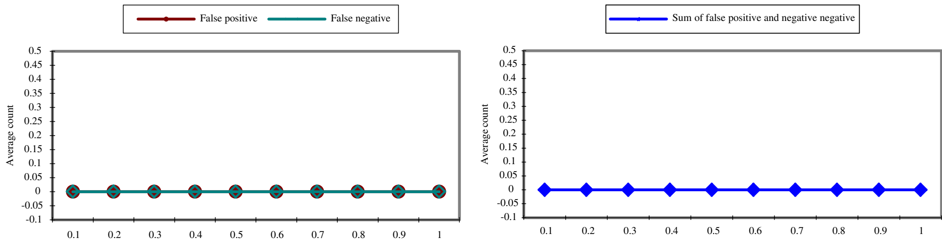
Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.



(a)



(b)



(c)

Fig. 14. Distribution of average goodness with different values of TFV. The pairs of plots (a), (b) and (c) show distribution of average goodness with different values of TF_i, TF_f and TF_t respectively. For each pair of plots, the left one shows the distribution of average false positive and negative, and the right one shows that of the sum of them. It must be noted that, the curves for average false positive and negative may overlap with each other. Additionally, a list of false positives and negatives of all the classes can be obtained at the URL http://www.ispecminer.com.

proper value of TFV should generate perfect CIMs with fewer false positives and negatives. In the experiment, we counted false positives and negatives manually according to JDK documentation.

Results of the experiment are shown in Fig. 14. The pairs of plots (a), (b) and (c) show distribution of average goodness with different values of TF_i, TF_f and TF_t respectively. From (a), we can see that the average count of false negatives is zero with any value of TF_i. However, false positives are introduced with the growth of TF_i. In order to obtain perfect CIMs, a small value of TF_i (lower than 0.4) should be

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

used. The scenario with TF_f is different from that of TF_i. As we can see from (b), the average count of false positives remains zero with the growth of TF_f, while low values of TF_f will result in false negatives. It indicates that a large value of TF_f (greater than 0.8) should be used in transformation. As to TF_t, we can see from (c) that both the average count of false positives and negatives remain zero with the growth of TF_t. Therefore, any value of TF_t may generate perfect CIMs. The above results provide guidance for us to set TFV to proper values. In addition, it demonstrates that our method can deal with noise effectively. As we can see, the maximum average counts of false positives and negatives are 0.4 and 0.7 respectively, which are trivial and acceptable. Above all, if more OUSs are used for learning, better results can be even achieved. Another finding from the experiment is that our method can deal with initial noise and transition noise better than final noise. The reason for this may be that more final noise exists in MCFs, which are mainly caused by accidental interruption of running applications. Finally, we investigated the connectivity of all the generated CIMs and found that all of them are connected. It indicates that our strategy of computing thresholds can guarantee connectivity of mined models.

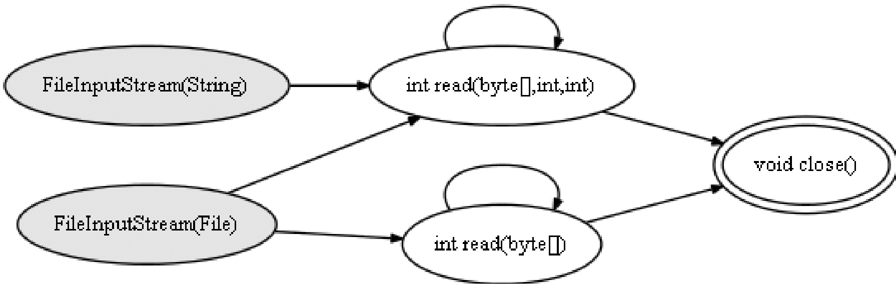
In conclusion, like any empirical evaluation, our experiment is limited in scope and the results may not generalize. However, we mined MCFs from real-world applications and investigated CIMs of 10 JDK classes, which were generated under a set of TFV from 0.1 to 1.0. Therefore, although further evaluation is needed, we believe that our results are promising. They show that our method can deal with noise effectively and guarantee connectivity of mined models. In addition, results of the experiment provide guidance for us to set TFV to proper values, with which, perfect CIMs can be obtained.

6.4. *Experiment 2: Inspecting mined class temporal specifications*

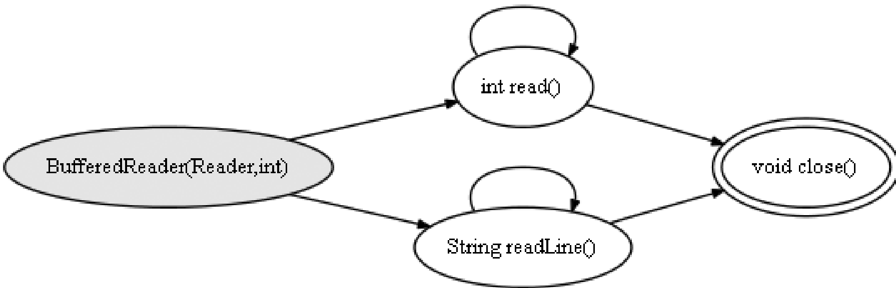
In this experiment, we examined the mined class temporal specifications against JDK documentations. Our purpose is to investigate how effective of our approach to obtain useful specifications. We first transformed the MCFs mined in the first experiment to CIMs based on a TFV (0.2, 0.9, 0.5), which were computed according to results of the first experiment. Then, we checked all the CIMs of 10 JDK classes against documentations. Several of the CIMs are shown in Fig. 15.

Take the CIM of class `java.io.FileInputStream` shown in Fig. 15(a) as an example. From the CIM, we can see that class `FileInputStream` has two beginning methods and one end method respectively. In order to use the class, we should first call method `FileInputStream(String)` or `FileInputStream(File)`. Next, methods `int read(byte[ ], int, int)` or `int read(byte[ ])` can be called multiple times to read bytes from the input stream. Finally, method `close()` should be used to close the stream and release resources. As we investigated, all the above temporal relationships are consistent with documentations. What should be noted is that, class `FileInputStream` possesses twelve public methods, while the CIM only includes five of them. This issue of partial specifications can be mitigated

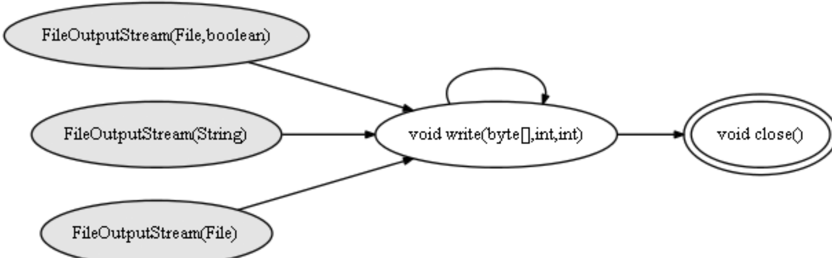
Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.



(a)



(b)



(c)

Fig. 15. Several of CIMs that we investigated. (a): CIM of class `java.io.FileInputStream`; (b): CIM of class `java.io.BufferedReader`; (c) CIM of class `java.io.FileOutputStream`. Ellipses filled with grey color are beginning methods and those denoted graphically by a double ellipse are end methods.

by providing more OUSs for learning. We examined all the CIMs and obtained similar results.

In conclusion, although we inspected mined specifications manually and the above results were preliminary in nature, we examined CIMs of 10 JDK classes and found that most of them were consistent with specifications documented, except for a small

number of false positives and negatives that we discussed in the first experiment. Therefore, we believe that ISpecMiner is able to mined useful specifications.

6.5. *Limitations of our approach*

Although our technique is able to mine useful class temporal specifications, it has the following limitations:

- Like any dynamic technique, effect of our approach largely depends upon the quality of generated test cases, which is a long-term conundrum in software testing. In this paper, we performed experiments using manual input data, which may bias the mined specifications. To our relief, this problem can be alleviated to some extent using our online learning algorithm, which can continuously refine existing MCFs.
- Just as any technique leveraging instrumentation, performance is a great concern. As our investigation, the applications instrumented by ISpecMiner run three to five times slower than the uninstrumented ones.
- For some technical reasons, ISpecMiner is unable to instrument Java native methods and several JDK classes.
- MCF breaks some properties of Markov chain, such as the sum of outgoing transition probabilities should be one. This makes MCF inconsistent with Markov chain and may limit the scope of its application.

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

7. Related Work

Class temporal specification is also referred to as component interface, object behavior model, object usage model, etc. Techniques of mining this kind of specification automatically have been studied extensively in recent years. Generally, these techniques can be divided into two categories: (1) static analysis based approach and (2) dynamic analysis based approach.

Static analysis based approach takes program source code or bytecode as input and infers specifications using techniques from data mining, machine learning, abstract interpretation, model checking, and etc. Since it does not require running applications with test cases, it can be highly automated and efficient. For instance, Wasylkowski [29, 30] proposed to mine object usage models from Java bytecode and a tool JADET was developed. However, it is sticky for static analysis based approach to deal with infeasible paths, complicated data structures and pointer alias.

Dynamic analysis based approach does not require program source code as input. It mines specifications from program execution traces, which can be obtained by running applications. This makes it a more dominant alternative in the case when source code is unavailable. To date, several tools of this kind have been developed, such as Daikon and ADABU. Both of these tools work in a similar manner. First a tracer is applied to collect program execution traces in a data file and then specifications are synthesized from the file. The difference between them is that Daikon

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

aims to learn invariants among variables and ADABU mine temporal specifications among methods which are most similar to ours. It has been proved that these tools are capable of learning many useful specifications. However, effect of this approach largely depends upon quality of generated test cases, which is a long time conundrum in software testing.

Whatever technique, a model should be employed to describe class temporal specification. In the past decade, most researchers perform the task using finite state automaton (FSA) and reduce the problem of mining class temporal specification to the well-known grammar inference problem. For instance, Lorenzoli [31] modeled class temporal specification using EFSM which extends from FSM and an automata learning algorithm GK-tail was adopted to infer the EFSM model of components. Alur [32] synthesized FSA model of class temporal specification using L* learning algorithms combined with model checking and abstract interpretation techniques.

Observed that FSA is a kind of deterministic model with inability to tolerate noise, Ammons*et al.* 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, they first infer PFSA from scenario strings using an off-the-shelf PFSA learner. Next, they transform PFSA to NFA by discarding rarely-used edges and weights. Finally, the NFA obtained is used for program verification and manual inspection. Our work is most similar to theirs. The differences are as follows: (1) our approach aims at object-oriented programs. In the work of Ammons, before learning specifications, flow dependence annotation should be used to refine program execution traces into interaction scenarios. In object-oriented programs, an interaction scenario is a method call sequence upon an object, which can be extracted directly from applications. To the best of our knowledge, this is the first work of mining class temporal specifications from object-oriented programs based on probabilistic models; (2) we learn class temporal specifications using an online algorithm. Relying on this algorithm, more universal specifications can be achieved by refining existing probabilistic models continuously; and (3) both of the works require transforming probabilistic models into deterministic models. However, details about the topic are omitted in their work. In this paper, we discussed connectivity problem of transformed models and an algorithm was proposed to resolve it.

8. Conclusions and Future Work

In this paper, we proposed to mine class temporal specifications dynamically relying on a probabilistic model extending from Markov chain. To the best of our knowledge, this is the first work of mining program specifications from object-oriented programs based on probabilistic models. Compared with similar works which apply probabilistic models to non-object-oriented programs, our technique does not require

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

annotating programs. In addition, we learn probabilistic models using an online algorithm, which enables our approach to infer more universal specifications. Realized that an unconnected model was an invalid specification, we discussed the connectivity problem of mined models and an algorithm was proposed to compute the maximum connected transition threshold, with which, connected models can be obtained. Although we use MCF to model class temporal specifications, our technique can be generalized to libraries and frameworks, where a specification includes methods from different classes.

We implemented our technique in a prototype tool `ISpecMiner` and used it to conduct several experiments. In evaluation, we mined models from several real-world applications and examined models of 10 JDK classes. Results of the experiments show that our approach can deal with noise effectively and obtain useful class temporal specifications. In addition, the proposed algorithm of computing maximum connected transition threshold can guarantee connectivity of mined models.

Markov chain is a traditional approach to handle sequential data. Some of its fundamentals may be applicable for the domain of mining program specifications. We leave this exploration to future work. In evaluation, we inspected mined program specifications manually. In the future, we will validate them in a more objective way by using them in program verification. Additionally, a comparison of our prototype tool `ISpecMiner` with others may be relevant. Due to some technical reasons, we leave it as an extension of this work.

Acknowledgments

This research is supported by Shenzhen Key Laboratory for High Performance Data Mining with Shenzhen New Industry Development Fund under grant No. CXB201005250021A and Natural Science Foundation of Hubei Province under grant No. 2014CFB1006.

References

1. M. Pradel and T. R. Gross, Automatic generation of object usage specifications from large method traces, in *ASE*, 2009, pp. 371–382.
2. M. Pradel and T. R. Gross, Leveraging test generation and specification mining for automated bug detection without false positives, in *ICSE*, 2012.
3. M. K. Ramanathan, A. Grama and S. Jagannathan, Static specification inference using predicate mining, *SIGPLAN Not.* **42**(6) (2007) 123–134.
4. S. Shoham and E. Yahav *et al.*, Static specification mining using automata-based abstractions, in *Proceedings of the International Symposium on Software Testing and Analysis*, 2007.
5. M. Di Penta, L. Cerulo and L. Aversano, The life and death of statically detected vulnerabilities: An empirical study, *Information and Software Technology* **51**(10) (2009) 1469–1484.
6. S. Thummalapenta and T. Xie, Alattin: Mining alternative patterns for defect detection, *Automated Software Engineering* **18**(3-4SI) (2011) 293–323.

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.

7. D. Lo and G. Ramalingam *et al.*, Mining quantified temporal rules: Formalism, algorithms, and evaluation, *Science of Computer Programming* **77**(6SI) (2012) 743–759.
8. M. D. Ernst and J. Cockrell *et al.*, Dynamically discovering likely program invariants to support program evolution, *IEEE Trans. Software Engineering* **27**(2) (2001) 99–123.
9. M. Gabel and Z. Su, Javert: Fully automatic mining of general temporal properties from dynamic traces, in *Proc. 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008.
10. J. Yang and D. Evans, Dynamically inferring temporal properties, in *Proc. 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2004.
11. J. H. Perkins and M. D. Ernst, Efficient incremental algorithms for dynamic detection of likely invariants, *SIGSOFT Softw. Eng. Notes*, **29**(6) (2004) 23–32.
12. 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.
13. G. Ammons, R. Bodik and J. R. Larus, Mining Specifications, *SIGPLAN Not.* **37**(1) (2002) 4–16.
14. O. Rasanen and U. K. Laine, A method for noise-robust context-aware pattern discovery and recognition from categorical sequences, *Pattern Recognition*, **45**(1) (2012) 606–616.
15. M. Casar and J. A. R. Fonollosa, Analysis of HMM temporal evolution for automatic speech recognition and verification, in *Proc. 9th International Conference on Text, Speech and Dialogue*, 2006.
16. E. A. Rua and J. Castro, Online signature verification based on generative models, *IEEE Transactions on Systems Man and Cybernetics Part B-Cybernetics*, **42**(4SI) (2012) 1231–1242.
17. D. Impedovo and G. Pirlo, Automatic signature verification: The state of the art, *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews* **38**(5) (2008) 609–635.
18. J. Fierrez and J. Ortega-Garcia *et al.*, HMM based online signature verification: Feature extraction and signature modeling, *Pattern Recogn. Lett.* **28**(16) (2007) 2325–2334.
19. M. M. Yin and J. Wang, Application of hidden Markov models to biological data mining: A case study, in *Data Mining and Knowledge Discovery: Theory, Tools, and Technology II* (2000), pp. 352–358.
20. M. D. Ernst and J. H. Perkins *et al.*, The Daikon system for dynamic detection of likely invariants, *Science of Computer Programming* **69**(1–3) (2007) 35–45.
21. V. Dallmeier and C. Lindig *et al.*, Mining object behavior with ADABU, in *Proc. International Workshop on Dynamic Systems Analysis*, 2006.
22. L. R. PARINER, A tutorial on hidden Markov models and selected application in speech recognition, in *Proc. IEEE* **77**(2) (1989) 257–286.
23. Javassist, 2013. http://en.wikipedia.org/wiki/Javassist.
24. Javassist, 2013. http://www.jboss.org/javassist.
25. S. Chiba and M. Nishizawa, An easy-to-use toolkit for efficient Java bytecode translators, in *Proc. 2nd International Conference on Generative Programming and Component Engineering*, 2003.
26. M. Tatsubori and T. Sasaki *et al.*, A bytecode translator for distributed execution of "legacy" Java software, in *Proc. 15th European Conference on Object-Oriented Programming*, 2001.
27. BCEL, 2013. http://commons.apache.org/proper/commons-bcel.
28. ASM, 2013. http://asm.ow2.org.

29. A. Wasylkowski, A. Zeller and C. Lindig, Detecting object usage anomalies, in *Proc. 6th Joint Meeting of the European Software Engineering Conference & ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2007*.
30. A. Wasylkowski, Mining object usage models, in *Companion to the Proceedings of the 29th International Conference on Software Engineering*, 2007.
31. D. Lorenzoli, L. Mariani and M. Pezz, Automatic generation of software behavioral models, in *Proc. 30th International Conference on Software Engineering*, 2008.
32. R. Alur and Pavol *et al.*, Synthesis of interface specifications for Java classes, *SIGPLAN Not.* **40**(1) (2005) 98–109.

Int. J. Soft. Eng. Knowl. Eng. 2015.25:573-604. Downloaded from www.worldscientific.com
by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY on 03/15/16. For personal use only.