Mining Parametric State-Based Specifications from Executions

Feng Chen and Grigore Roşu
University of Illinois at Urbana-Chamapign
Urbana, Illinois, USA 61801
fengchen, grosu@illinois.edu

Abstract

This paper presents an approach to mine parametric state-based specifications from execution traces, which can involve multiple components. We first discuss a general framework for mining parametric properties from execution traces, which allows one to apply non-parametric mining algorithms to infer parametric specifications without any modification. Then we propose a novel mining algorithm that extends the Probabilistic Finite State Automata (PFSA) approach to infer finite automata that describe system behaviors concisely and precisely from successful executions. The presented technique has been implemented in a prototype tool for Java, called jMiner, which has been applied to a number of real-life programs, including Java library classes and popular open source packages. Our experiments generated many meaningful specifications and revealed problematic behaviors in some programs, showing the effectiveness of our approach.

1 Introduction

Automatically mining properties, e.g., API patterns and usage scenarios, from observed execution traces can be used in many stages of software development, e.g., program understanding, program analysis, error detection and testing. In this paper, we propose a novel technique to learn parametric state-based specifications from execution traces, which can be used to specify behavioral properties involving multiple components in the program. Parametric specifications are specifications with free variables, called parameters, that need to be instantiated at runtime. Figure 1 shows an example of parametric specification that was inferred by the technique proposed in this paper. The specification contains a deterministic finite automaton (DFA) and an equivalent regular pattern. It describes a usage pattern involving two Java Util library classes, namely, Collection and Iterator. Collection is the base class to implement collection-typed data structures, e.g., lists and sets. Iterator is an interface used to enumerate elements in a Collection object. In Figure 1, update is an event representing a function call on a Collection object that changes the content of the object, such

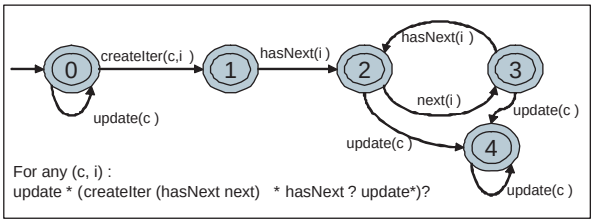

Figure 1. Collection-Iterator **usage pattern**

as add and remove. It has one parameter, namely, the target Collection object. createIter is the event for the creation of an Iterator instance from a Collection object and it has two parameters: the underlying Collection object and the created Iterator instance. hasNext and next represent invocations on two major functions of Iterator. The former checks whether the iterator has more elements to enumerate and the latter fetches the next element in the iterator.

The specification in Figure 1 captures two safety properties. First, once the automaton leaves states 2 and 3 and enters state 4, no usage of the iterator can be seen afterwards (state 4 does not have any edge back). It states the following safety property: if an iterator i is created for a collection c, the contents of c should not be changed while i is being used. A violation of this property will cause an exception to be thrown from Iterator. Second, Figure 1 also shows a pattern of using an Iterator object, that is, start with a hasNext and then call functions next and hasNext alternatively. In other words, every call to the next function should be guarded by a hasNext function call. The latter is not required by the Java API specification, but a violation of this constraint may imply unsafe uses of Iterator. In fact, our tool mined a conflicting pattern of using Iterator from the pmd benchmark in the Dacapo [3] suite, indicating a potential bug in the program, which is discussed in Section 4.

Our approach mines parametric specifications from parametric traces, i.e., traces containing events with parameter bindings. A parametric trace is usually comprised of many meaningful traces merged, which may lead to confusions during the learning process and significantly increase the difficulty of inferring meaningful specifications. Figure 2 shows a fragment of a parametric trace that was

```
...
update(Collection:158)
createIter(Collection:158, Iterator:119)
hasNext(Iterator:119)
next(Iterator:119)
update(Collection:148)
hasNext(Iterator:119)
next(Iterator:119)
update(Collection:148)
hasNext(Iterator:119)
next(Iterator:119)
update(Collection:148)
hasNext(Iterator:119)
createIter(Collection:263, Iterator:131)
hasNext(Iterator:131)
...
```

Figure 2. Fragment of a logged trace

logged in our experiments to infer the specification in Figure 1. Every event comes with parameter bindings, for example, the first event, update(Collection:158), instantiates the parameter Collection with a concrete object represented using the first three digits of its runtime hashcode[1], 158. When observing a running program, operations of different Collection objects, e.g., update(Collection:158) and update(Collection:148), and operations of different Iterator objects, e.g., next(Iterator:119) and hasNext(Iterator:131) are mixed, even though not all of them are interrelated. The original trace, from which Figure 2 was extracted, contains tens of thousands of events with many different parameter bindings, making it highly non-trivial to infer useful specifications from the trace efficiently.

Our approach is composed of a general framework for parametric property mining, which allows one to apply non-parametric mining algorithms to infer parametric specifications without any modification, and a mining algorithm that infers DFAs from non-parametric traces. The framework, as depicted in Figure 3, uses aspect-oriented programming (AOP) [15] to specify and instrument observation points into the base program to collect execution traces with parameter information. We then employ a parametric trace slicing algorithm to slice the collected trace into non-parametric trace slices according to different sets of parameters. The non-parametric slices are passed to the mining component to mine (non-parametric) specifications. At the end, we associate the inferred specification with parameter information to produce a parametric specification as output.

The major contributions of this paper are:

1. We proposed a parametric specification mining framework based on AOP and parametric trace slicing, allowing one to use non-parametric mining techniques on parametric traces.
2. We developed a mining algorithm to infer state-based specifications from correct execution traces. It extends the Probabilistic Finite State Automata (PFSA) [19] algorithm to achieve more accurate results. The algorithm outputs DFAs and equivalent regular patterns, which describe the system behavior in a compact and human-readable way.
3. We implemented the proposed approach in a tool prototype for Java, named jMiner, and applied it to many programs. The results show that our approach is effective in mining compact and accurate parametric specifications. We also detected a few problematic behaviors in the Dacapo benchmark using jMiner.

[1]The full hashcode is used in implementation.

2 Parametric Mining Framework

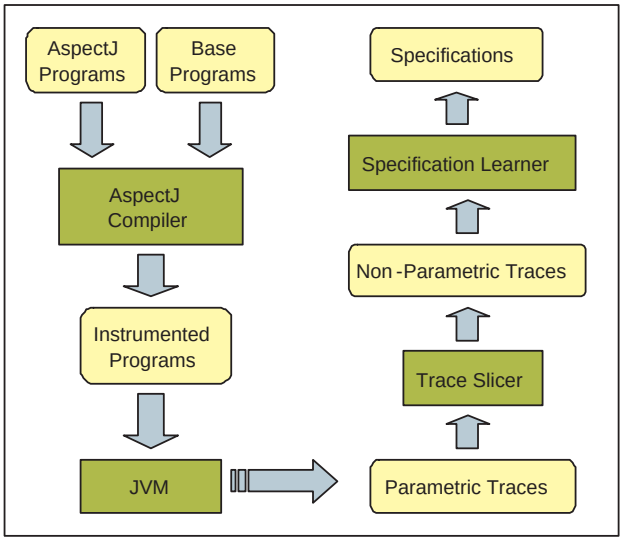

Figure 3. Parametric Specification Mining

Our solution to learn parametric state-based properties is composed of three stages as depicted in Figure 3. The first stage is to collect execution traces that exercise the uses of the components under learning. A set of base programs are needed to generate the traces. For now we simply assume that the base program is given. In practice, one may use existing test cases, benchmarks (for example, we used the Dacapo benchmark [3] in our experiments) or some automated tools, such as the model-checker-based approach proposed in [1], as the base programs. The base program is instrumented to log its uses of the target components. We make use of aspect-oriented programming (AOP) languages, e.g., AspectJ [14] for Java, in our approach to describe the desired observation points in the based program, as well as to instrument the base program with logging instructions in the designated observation points. AspectJ provides an expressive, pattern-based language to describe *join points* in a program, i.e., those points where one may insert actions, giving us an elegant way to capture the uses of certain components. Figure 4 shows the AspectJ code used to log method invocations of Collection and Iterator objects, which generated the trace in Figure 2 in our experiments. The AspectJ compiler then takes the AspectJ program and the base program as input and generates an instrumented program that, when

```
import java.util.*;
public aspect col_iter_Logger {
TraceWriter w=new TraceWriter("col-iter.log");

after(Collection c):
(call(* Collection+.add*(..))||call(* Collection+.clear())
 || call(* Collection+.remove*(..))) && target(c){
w.log("update", new Object[]{c});
}
after(Collection c) returning(Iterator i):
( call(* Collection+.iterator()) && target(c)) {
w.log("createIter", new Object[] {c, i});
}
after(Iterator i):
( call(* Iterator+.next()) && target(i)) {
w.log("next", new Object[] {i});
}
after(Iterator i): (
 call(* Iterator+.hasNext()) && target(i)) {
w.log("hasNext", new Object[] {i});
}
}
```

Figure 4. ApsectJ code for logging uses of Collection **and** Iterator

executed, logs a projection of its execution at the specified join points.

Our framework adopts a divide-and-conquer solution, that is, the logged parametric trace is first sliced according to a set of parameters for which one is interested in generating specifications, e.g., {Collection, Iterator} [2]. The trace slicer, discussed in detail in Section 2.1, will produce from the logged parametric trace a set of non-parametric traces, each of which contains all the events compatible with a specific binding of the designated parameter set, e.g., in the above example, a specific pair of Collection and Iterator objects. The generated non-parametric traces are used as input to the specification miner, which mines non-parametric specifications. The mined specification is then associated with parameter information to output a parametric specification. This way, the miner can focus on the inferring process without worrying about parameters, allowing reuses of existing techniques and reducing complexity of developing new algorithms. The mining algorithm discussed in Section 3 shows an effective application within our framework, and we believe that other techniques, such as the PCFO mining in [1], can also be incorporated into our framework to infer different types of parametric specifications.

2.1 Slicing Traces by Parameters

Slicing a parametric trace according to a set of parameters is based on a process of dispatching events to trace slices corresponding to different binding instances of the parameter set. Although the intuition is clear, developing a correct slicing algorithm is non-trivial, considering the fact that an event may contain an incomplete binding of the given parameter set and/or irrelevant parameter instances.

[2]The desired parameter set is not necessary the same as the classes/interfaces under observation, for example, one may choose a parameter set to be {Collection, Collection} in order to infer the specification of interactions between different Collection objects.

For example, in Figure 2, if we choose {Collection} as the target parameter set, a createIter event contains, in addition to a desired Collection instance, an irrelevant Iterator instance. If we choose {Collection, Iterator} as the target parameter set, an update event contains only a Collection instance, leaving the Iterator parameter unbound. An important aspect here is that one does *not* want to traverse the log multiple times for each parameter instance because that would be extremely inefficient when the log is large. We next formalize the parametric slicing process and introduce an efficient slicing algorithm based on the work proposed in [22].

Definition 1 *Let $\mathcal{E}$ be a set of (non-parametric) events, called **base events** or simply **events**. An $\mathcal{E}$-**trace**, or simply a (non-parametric) **trace**, is any finite sequence of events in $\mathcal{E}$, that is, an element in $\mathcal{E}^*$. If event $e \in \mathcal{E}$ appears in trace $w \in \mathcal{E}^*$ then we write $e \in w$.*

Let $[A \to B]$ and $[A \stackrel{\circ}{\to} B]$ denote the sets of total and respectively partial functions from A to B. What follows extends the definition above to parametric events and traces.

Definition 2 ***(Parametric events and traces)*** *Let X be a set of **parameters** and let V_X be a set of corresponding **parameter values**. If $\mathcal{E}$ is a set of base events like in Definition 1, then let $\mathcal{E}\langle X\rangle$ denote the set of corresponding **parametric events** $e\langle\theta\rangle$, where e is a base event in $\mathcal{E}$ and θ is a partial function in $[X \stackrel{\circ}{\to} V_X]$. θ is called a **parameter instance** or **parameter binding**. A **parametric trace** is a trace with events in $\mathcal{E}\langle X\rangle$, that is, a word in $\mathcal{E}\langle X\rangle^*$.*

For the trace in Figure 2, we have $\mathcal{E}$ = {update, createIter, next, hasNext}, X = {Collection, Iterator} and V_X = {158, 119, ...}. update(Collection:158) and next(Iterator:119) are simplified representations of update$\langle\theta$(Collection)=158$\rangle$ and next$\langle\theta$(Iterator)=119$\rangle$, respectively.

Definition 3 *Two parameter instances θ and θ' are **compatible** iff for any $x \in \mathit{Dom}(\theta) \cap \mathit{Dom}(\theta')$, $\theta(x) = \theta'(x)$.*

For example, (Collection:158) is compatible with (Collection:158, Iterator:119).

Given a parameter set, some events may contain more parameter instances than necessary. The following definition removes irrelevant parameters from an event.

Definition 4 ***(Parameter restriction)*** *Given partial function $\theta \in [X \stackrel{\circ}{\to} V_X]$ and parameter set Y, we let $\theta\restriction_Y$ be the **restriction** of θ over Y defined as follows: $\theta\restriction_Y(x) = \theta(x)$ if $x \in X \cap Y$. Also, we say θ' is **less informative** than θ, denoted as $\theta' \sqsubseteq \theta$, iff for any $x \in X$, if $\theta'(x)$ is defined then $\theta(x)$ is also defined and $\theta'(x) = \theta(x)$.*

For example, suppose Y = {Collection}, then we have (Collection:158)$\restriction_Y$= (Collection:158), (Iterator:119)$\restriction_Y$ = $\perp$ where $\perp$ means the empty function, and (Collection:263, Iterator:131)$\restriction_Y$ = (Collection:263).

Definition 5 ***(Restricted trace slicing)*** *Given parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$ and partial function θ in $[Y \stackrel{\circ}{\rightarrow} V_X]$ s.t. $Y \subseteq X$, we let the* ***restricted θ-trace slice*** *$\tau \upharpoonright_\theta \in \mathcal{E}^*$ be the non-parametric trace in $\mathcal{E}^*$ defined as follows:*

- $\epsilon \upharpoonright_\theta = \epsilon$, *where ϵ is the empty trace/word, and*
- $(\tau\, e\langle\theta'\rangle) \upharpoonright_\theta = \begin{cases} (\tau \upharpoonright_\theta)e & \text{when } \theta' \lfloor_Y \sqsubseteq \theta \\ \tau \upharpoonright_\theta & \text{when } \theta' \lfloor_Y \not\sqsubseteq \theta \end{cases}$

Intuitively, a restricted trace slice $\tau \upharpoonright_\theta$ first filters out all the parametric events that contain parameter instances incompatible with θ or no θ-compatible parameter instances at all. Then, for the events relevant to θ, it forgets the parameters to build a non-parametric trace. This way, the restricted trace slicing can be regarded as the combination of restriction and the trace slicing defined in [22]. In the rest of this paper, we use "trace slicing" to refer to "restricted trace slicing" for simplicity. For parameter instance ⟨Iterator:119⟩ in Figure 2, the trace slice is:

```
createIter(Collection:158, Iterator:119)
hasNext(Iterator:119)
next(Iterator:119)
hasNext(Iterator:119)
next(Iterator:119)
hasNext(Iterator:119)
next(Iterator:119)
hasNext(Iterator:119)
```

Other events are either incompatible with ⟨Iterator:119⟩, e.g., hasNext⟨Iterator:131⟩ binds Iterator with a different value, or have no compatible instance, e.g., add⟨Collection:158⟩ provides no value for Iterator. Therefore the corresponding sliced trace is createIter hasNext next hasNext next hasNext next hasNext.

A simple algorithm to slice parametric traces according to a given parameter set X can be immediately implemented following Definition 5. First, we go through parametric trace τ to construct all possible parameter instances θ with $\mathsf{Dom}(\theta) = X$. Then τ is scanned again to distribute every event to $\tau \upharpoonright_\theta$ for different τ according to the condition in Definition 5. However, there are two drawbacks of this algorithm. First, it is not efficient because τ will be scanned at least twice during slicing, resulting in a much slower slicing process when τ is very long. Second, and more important, it assumes an offline slicing mode, i.e., slicing the logged trace only after the program terminates. Although it is a reasonable restriction, we may still need to slice traces for programs with infinite executions, e.g., webservers.

Figure 5 shows an efficient slicing algorithm, called $\mathbb{S}\langle X\rangle$ where X is the set of target parameters, which constructs parameter instances and distributes events at the same time, allowing slicing to be carried out along with the execution of the base program and supporting potentially infinite executions. In $\mathbb{S}\langle X\rangle$, there are two global variables, namely, Δ that maps a parameter instance into the corresponding trace slice and $\mathcal{U}$ that maps parameter instance θ into the set of θ' that has been constructed and more informative than θ. We omit the main function of algorithm

```
Algorithm 𝕊⟨X⟩
Globals: mapping Δ : [[X →° V_X] →° Σ*] and
         mapping 𝒰 : [X →° V_X] → 𝒫_f([X →° V_X]) and
Initialization: 𝒰(θ) ← ∅ for any θ ∈ [X →° V_X], Δ(⊥) ← ε

function handleEvent(e⟨θ_e⟩)
 1 θ ← θ_e ⌊_X
 2 if (θ = ⊥) then return endif
 3 if Δ(θ) undefined then
 4 ⋮ foreach θ_max ⊏ θ (in reversed topological order) do
 5 ⋮ ⋮ if Δ(θ_max) defined then goto 7 endif
 6 ⋮ endfor
 7 ⋮ defineTo(θ, θ_max)
 8 ⋮ foreach θ_max ⊏ θ (in reversed topological order) do
 9 ⋮ ⋮ foreach θ_comp ∈ 𝒰(θ_max) compatible with θ do
10 ⋮ ⋮ ⋮ if Δ(θ_comp ⊔ θ) undefined then
11 ⋮ ⋮ ⋮ ⋮ defineTo(θ_comp ⊔ θ, θ_comp)
12 ⋮ ⋮ ⋮ endif
13 ⋮ ⋮ endfor
14 ⋮ endfor
15 endif
16 foreach θ' ∈ {θ} ∪ 𝒰(θ) do
17 ⋮ Δ(θ') ← Δ(θ')e
18 endfor

function defineTo(θ, θ')
1 Δ(θ) ← Δ(θ')
2 foreach θ'' ⊏ θ do
3 ⋮ 𝒰(θ'') ← 𝒰(θ'') ∪ {θ}
4 endfor
```

Figure 5. Parametric slicing algorithm $\mathbb{S}\langle X\rangle$

$\mathbb{S}\langle X\rangle$ here, which simply takes the input parametric trace τ and invokes handleEvent on each event in τ. The handleEvent function can be roughly divided into four steps: first, it restricts the parameter instance of the input event using the given parameter set X (line 1); second, if the event does not contain any relevant parameter instance, it is skipped (lines 2); third, otherwise, if the restricted parameter instance θ has not been encountered yet ($\Delta(\theta)$ has not been defined), we will construct new parameter instances based on θ and create corresponding trace slices (lines 3 to 15). fourth, the input event is appended to all the trace slices corresponding to θ' with $\theta \sqsubseteq \theta'$ (line 16 to 18).

In the third step above, we need to decide which new parameter instances to create and how to initialize the corresponding trace slices. First, we search for the most informative parameter instance θ_{max} that is less informative than θ and has been constructed (lines 4 to 6). θ_{max} is then used to initialize the trace slice created for θ (line 7). Then we search for all the parameter instances that are compatible with θ and have been constructed; such compatible parame-

ter instances are merged with θ to create new parameter instances when needed (lines 8 to 14). The search is achieved by making use of the mapping $\mathcal{U}$ and based on the following observation: if θ_{comp} is compatible with θ then it can be found in the list $\mathcal{U}(\theta_{max})$ for some θ_{max} less informative than θ and θ_{comp}. Therefore, the nested loop (lines 8 to 12) will find all $\theta-$compatible parameter instances and create new parameter instances and corresponding trace slices.

Creation and initialization of a new trace slice is done by the defineTo function, which takes two parameter instances, θ and θ', as input and uses the trace slice for θ' to create the trace slice for θ (line 1). The defineTo function also updates the mapping $\mathcal{U}$ by adding the newly defined θ into $\mathcal{U}(\theta'')$ for any θ'' less informative than θ (line 2 to 4). This way, we ensure that $\mathcal{U}$ always contains the complete information about constructed parameter instances. The correctness of Algorithm $\mathbb{S}\langle X\rangle$ is guaranteed by the following:

Theorem 1 *For any $\tau \in \mathcal{E}\langle X\rangle^*$, given parameter set Y s.t. $Y \subseteq X$, we have $\tau \downarrow_\theta = \Delta(\theta)$ for any $\theta \in [Y \xrightarrow{\circ} V_X]$ and $\mathsf{Dom}(\theta) = Y$ after we run algorithm $\mathbb{S}\langle Y\rangle$ on τ.*

Proof. Can be directly derived from the proof of algorithm $\mathbb{C}$ in [22]. □

Example. Table 1 illustrates a run of algorithm $\mathbb{S}\langle X\rangle$ over the first five events in Figure 2, given the parameter set $X = \{$Collection, Iterator$\}$. In this example run, when the first event, update(Collection:158), is received, a new trace slice is created for the parameter instance (Collection:158) and the event update is added to the slice. The second event createIter(Collection:158, Iterator:119) also results in a new trace slice which is initialized with the trace slice for (Collection:158) because (Collection:158) is less informative than (Collection:158, Iterator:119). Then the event createIter is appended to the new slice. A trace slice for (Iterator:119) is created at the third event hasNext(Iterator:119) with the initial value ϵ because no other parameter instance is less informative than it. The event hasNext is then appended to both slices corresponding to (Collection:158, Iterator:119) and (Iterator:119). The fourth event next(Iterator:119) does not lead to creation of new trace slice and hasNext is appended to both slices corresponding to (Collection:158, Iterator:119) and (Iterator:119). Two new trace slices are created at the fifth event update(Collection:148), namely, one for (Collection:148) and one for (Collection:148, Iterator:119). The former is initialized to be ϵ and the latter is initialized using the slice for (Iterator:119).

Discussions. Algorithm $\mathbb{S}\langle X\rangle$ also generates trace slices for parameter instance θ with $\mathsf{Dom}(\theta) \subset X$. In the example in Table 1, we have the slices for (Collection:158) and for (Iterator:119). It is worth noting that these slices are used for intermediate purposes only, e.g., the slice for (Collection:158) is used to initialize the slice for (Collection:158, Iterator:119) in Table 1. So they can be different from the resulting slice of $\mathbb{S}\langle X\rangle$ if $X = \mathsf{Dom}(\theta)$. For instance, if $X = \{$Iterator$\}$ then the slice for ((Iterator:119) will contain the first createIter in Figure 2, which is not included in the slice for (Iterator:119) in Table 1 when $X = \{$Collection, Iterator$\}$.

2.2 Removing Redundant Traces

One may also notice that one of the new parameter instances created at the fifth event in the above example, that is, (Collection:148, Iterator:119), is redundant since Iterator:119 is an iterator over Collection:158 and is not related to Collection:148 during execution. Redundant traces may bring noise into the learning process, reducing the precision of the resulting specification. We develop a heuristic to remove redundant trace slices, as described in what follows.

Definition 6 *For any $\tau \in \mathcal{E}\langle X\rangle^*$, we say two parameter instances θ and θ' are **connected** in τ iff either there exists an event $e\langle\theta''\rangle \in \tau$ s.t. $\theta \sqsubseteq \theta''$ and $\theta' \sqsubseteq \theta''$, or there exists a parameter instance θ'' s.t. θ and θ'' are connected in τ and θ' and θ'' are connected in τ.*

Intuitively, θ and θ' are connected if they interact through an event or a series of events in τ. For example, in Figure 2, (Collection:158) and (Iterator:119) are connected because of the first createIter event.

Definition 7 *For any $\tau \in \mathcal{E}\langle X\rangle^*$, parameter instance θ is **redundant** iff we can find two non-empty θ' and θ'' s.t. $\theta' \sqsubset \theta$, $\theta'' \sqsubset \theta$ and θ' and θ'' are not connected in τ.*

Only those trace slices whose corresponding parameter instances are not redundant are used in the mining process. For the example in Table 1, the trace slice for (Collection:148, Iterator:119) will not be used for specification mining because there is no event that connects Collection:148 with Iterator:119. The redundancy check of parameter instances can be efficiently implemented by storing parameter instances as graphs whose nodes are instances of individual parameters and edges connect nodes that are connected in τ. When the trace slice for θ is updated using event $e\langle\theta_e\rangle$, θ_e is used to add new edges in the graph for θ. Therefore, a parameter instance is redundant if its graph is not connected.

Since algorithm $\mathbb{S}\langle X\rangle$ can be used for online slicing, i.e., slicing the trace on the fly, one may directly generate non-parametric traces from the observed execution using $\mathbb{S}\langle X\rangle$. We choose to have a simple logging process and then slice the logged trace because it reduces the runtime overhead of observing the execution and allows reusing the logged trace for slicing using different parameter set. For example, one can use the trace in Figure 2 to generate non-parametric traces for $\{$Collection$\}$, $\{$Iterator$\}$ or $\{$Collection, Iterator$\}$.

3 Mining From Successful Traces

We develop a state-based specification mining algorithm that infers a deterministic finite automaton (DFA) together with an equivalent regular pattern from non-parametric

update(Collection:158)	createIter(Collection:158, Iterator:119)	hasNext(Iterator:119)
(): ϵ (Collection:158): update	(): ϵ (Collection:158): update (Collection:158, Iterator:119): update createIter	(): ϵ (Collection:158): update (Collection:158, Iterator:119): update createIter hasNext (Iterator:119): hasNext

next(Iterator:119)	update(Collection:148)	...
(): ϵ (Collection:158): update (Collection:158, Iterator:119): update createIter hasNext next (Iterator:119): hasNext next	(): ϵ (Collection:158): update (Collection:158, Iterator:119): update createIter hasNextnext (Iterator:119): hasNext next (Collection:148): update (Collection:148, Iterator:119): hasNext next update	... ...

Table 1. A run of the trace slicing algorithm $\mathbb{S}\langle X \rangle$ (top table first, followed by the bottom table).

traces. The proposed mining process is illustrated in Figure 6. It consists of three components, namely, a Probabilistic Finite State Automata(PFSA) learner, an automaton refiner and a regular pattern generator. The regular pattern generator uses the Brzozowski method [5] to generate equivalent regular patterns from deterministic finite state machines. The generated regular pattern is then simplified using a set of rules. The algorithm produces a reasonably compact (but not necessarily minimal) pattern that can be easily understood, facilitating documentation and program understanding. We next discuss the PFSA learner and the refiner in more depth.

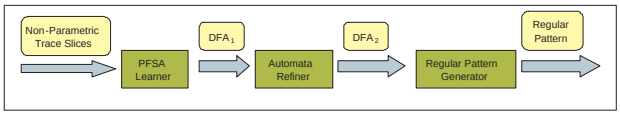

Figure 6. Approach overview.

3.1 PFSA Learner

A PFSA learner infers a finite state machine from a set of strings, i.e., non-parametric traces in this paper. The inferred state machine accepts at least the given set of strings and may allow more since the PFSA learner usually generalizes during the learning process. A number of approaches have been proposed to build the PFSA learner [8]. We adopted the sk-string algorithm in [19] because it performs better in inferring small automata. The interested reader should refer to [19] for a full description and explanation.

The sk-string PFSA learner first constructs a prefix tree, which is essentially an FSA that accepts precisely the input set of strings. Each arc of the prefix tree is labeled with a frequency that represents how many times the arc was traversed during the creation of the tree. The sk-string algorithm is then used to merge states in the prefix tree to build a more compact and more general non-deterministic finite automaton. State merging is based on the concept called *sk-equivalence*. Let Σ be the set of words used in the strings, Q be the set of states in the prefix tree, $\delta : Q \times \Sigma^* \rightarrow 2^Q$ be the transition function, and F_C be the set of final states. The set of k-strings of state q is then defined to the set $\{z \mid z \in \Sigma^*, \mid z \mid = k \wedge \delta(q, z) \subset Q \vee \mid z \mid < k \wedge \delta(q, z) \cap F_C \neq \emptyset\}$. Each k-string has a probability associated with it which is the product of the probabilities of the arcs traversed in generating the string. Two states are considered mergeable if the sets consisting of the top s percent of their distribution of k-string are the same (i.e., sk-equivalence). It is computed as follows: the k-stirngs of a state are arranged in decreasing order of their probabilities. The top n strings, whose probabilities add up to s percent or more with n being as small as possible, are retained and the remaining strings (those having lower probabilities) are ignored. Two states are sk-equivalent if the sets of the top n strings of both are the same. The process of merging states is repeated until no more states are sk-equivalent. This way, the algorithm infers an NFA accepting a superset of the input strings. Then it converts the NFA into a DFA.

In our approach, the set of input strings to the PFSA learner is the set of non-parametric trace slices generated from the logged trace by the trace slicer. The learner outputs a DFA, whose nodes represent the states of the involved components and edges are labeled with events. Figure 7 gives the DFA describing the interaction between a Collection object and a Iterator instance, which is inferred by the PFSA learner, together with an equivalent regular pattern.

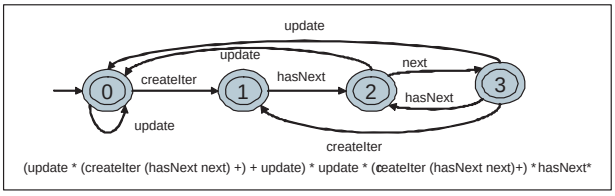

Figure 7. PFSA for Collection **and** Iterator

3.2 Automata Refiner

Although the PFSA learner usually generates a small automaton, the resulting automaton often over-generalizes and accepts a large number of traces that should not be allowed. For example, a trace createIter hasNext next createIter can be accepted by the automaton in Figure 7, while it is impossible to occur during any execution because for any given pair of Collection and Iterator objects, only one createIter event can be observed. An over-generalized specification undermines the effectiveness of its applications. For example, it may cause misunderstanding of the system behaviors when used for reverse engineering, or miss potential errors

when used for program testing and verification. Also, an over-generalized automaton often (but not necessarily) produces a more complicated regular pattern as the final result, e.g., the one in Figure 7, increasing the difficulty of human inspection. Therefore, we devised an algorithm to refine the PFSA-inferred automaton using again the traces from which the automaton is inferred, which is depicted in Figure 8.

Definition 8 ***Automaton.*** *An* automaton *is a tuple* $(S, \Sigma, i, \delta, F)$ *where* S *is a set of states,* Σ *is a set of events,* $i \in S$ *is the initial state,* $\delta : [S \times \Sigma \xrightarrow{\circ} S]$ *is the transition function, and* $F \subseteq S$ *is the set of final states.*

In algorithm $\mathbb{R}$, we first expand the input automaton using the expand function which splits each state according to its incoming edges. The incoming edges are counted as follows: if $\delta(s, e) = s'$ for some $s \neq s'$ then e represents an incoming edge to s'. Also, we assume that the initial state has a default incoming edge (lines 3 to 5 in expand). Hence, state 0 in Figure 7 has two incoming edges, namely, a default one and one from state 3, while state 1 also has two incoming edges from states 0 and 3. If state s has n incoming edges then n new states are generated for the new automaton and we keep the mapping from s to the corresponding set of newly created states in γ (lines 6 to 8 in expand). Function expand then builds the transition function for the new automaton (lines 10 to 23) as follows: if $\delta(s', e) = s$ is a transition in the input automaton and $s \neq s'$ then we choose a state s'' from $\gamma(s)$ with no incoming edges at this point and add transitions from every state in $\gamma(s')$ to s''. In addition, if s is a final state then all states in $\gamma(s)$ are also final; if s is the initial state then we choose a state from $\gamma(s)$ with no incoming edges as the new initial state. This way, we expand the input automaton to an equivalent automaton in which every state has a set of incoming edges corresponding to one incoming edge in the original automaton. Figure 9 shows the expanded automaton of Figure 7.

Algorithm $\mathbb{R}$ then traverses the expanded automaton using the input set of traces and marks the transitions used in the traversal (lines 3 to 13). After all the traces are applied, $\mathbb{R}$ removes the unmarked transitions from the expanded automaton. For example, Figure 10 shows the automaton after $\mathbb{R}$ traverses the expanded automaton in Figure 9. Two edges are removed in Figure 10, namely, from 0_1 to 1_1 and from 3 to 1_2 because they are not traversed in any trace. The reduced automaton is then compresses by merging states that have the same out-going transitions and removing those states that have no incoming states. For example, state 1_2 in Figure 10 needs to be removed because it has no incoming edges and states 2_1 and 2_2 should be merged. At the end, we associate the compressed automaton with parameter information removed during parametric slicing. This way, we can achieve the automaton shown in Figure 1. Note that the refined automaton may contain more states but it is more restrictive and precise than the PFSA-inferred automaton, and

```
Algorithm ℝ
Input: automaton A = (S, Σ, i, δ : [S × Σ →° S], F),
       set of traces T ⊂ Σ*
Output: automaton A_r
Locals: automaton A' = (S', Σ, i', δ', F'),
        state s, s' and transition function δ_r
 1  A' ← expand(A)
 2  δ_r ← ⊥
 3  foreach (τ ∈ T) do
 4  ⋮  s ← i'
 5  ⋮  foreach e ∈ τ do
 6  ⋮  ⋮  s' ← s
 7  ⋮  ⋮  s ← δ'(s, e)
 8  ⋮  ⋮  δ_r(s', e) ← s
 9  ⋮  ⋮  if (δ_r = δ') then
10  ⋮  ⋮  ⋮  goto 14
11  ⋮  ⋮  endif
12  ⋮  endfor
13  endfor
14  A' ← (S', Σ, i', δ_r, F')
15  A_r ← mergeIdenticalStates(A')
16  output(A_r)

function expand(A = (S, Σ, i, δ, F))
Output: automaton A' = (S', Σ, i', δ', F')
Locals: integer n, set of states D, mapping γ : S → 2^S'
Initialization: S' ← ∅, F' ← ∅, δ' ← ⊥
 1  foreach (s ∈ S) do
 2  ⋮  n ← countIncomingEdges(s, A)
 3  ⋮  if (s = i) do
 4  ⋮  ⋮  n ← n + 1
 5  ⋮  endif
 6  ⋮  D ← getFreshStates(n)
 7  ⋮  S' ← D ∪ S'
 8  ⋮  γ(s) ← D
 9  endfor
10  foreach (s ∈ S) do
11  ⋮  foreach (s' ≠ s ∈ S s.t. δ(s', e) = s for some e) do
12  ⋮  ⋮  s'' ← pickOneWithNoIncomingEdge(γ(s), δ')
13  ⋮  ⋮  foreach (s''' ∈ γ(s')) do
14  ⋮  ⋮  ⋮  δ'(s''', e) = s''
15  ⋮  ⋮  endfor
16  ⋮  endfor
17  ⋮  if (s ∈ F) then
18  ⋮  ⋮  F' ← F' ∪ γ(s)
19  ⋮  endif
20  ⋮  if (s = i) then
21  ⋮  ⋮  i' ← pickOneWithNoIncomingEdge(γ(s), δ')
22  ⋮  endif
23  endfor
24  return A'
```

Figure 8. Automaton refining algorithm $\mathbb{R}$

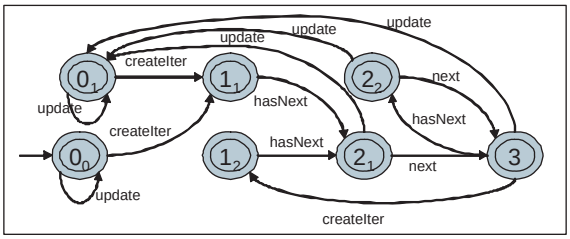

Figure 9. Expanded automaton of Figure 7

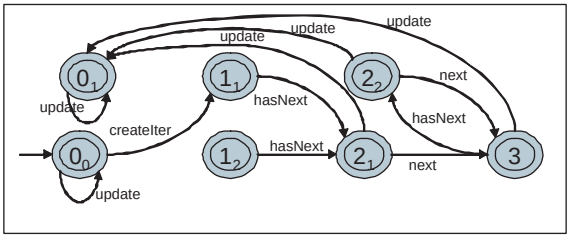

Figure 10. Reduced automaton of Figure 9

still accepts all the input traces. Moreover, our experiments show that the refined automaton usually results in a much more compact regular pattern like in the above example.

3.3 Limitations and Heuristics

One major limitation of our mining technique is that only positive traces are considered, i.e., all the input traces are regarded as "correct" traces. In other words, the learning process cannot use "negative" traces to refine and generalize the inferred automaton. Therefore, it requires more input traces than the techniques that make use of both positive and negative traces. We choose this algorithm in our approach because it is easier to collect positive traces than to collect negative traces from existing, well-used programs, especially when no a priori knowledge is given for the property we want to infer. Also, our experiments show that the number of traces needed for this algorithm to produce meaningful results is usually reasonable: in fact, no special effort was spent to obtain enough traces in our experiments, partially because the parametric slicing technique produces many non-parametric slices from a few observed parametric traces. Moreover, even when the input contains incorrect traces, it is easy for the user to catch the abnormal behaviors using the human-readable output, such as in the problematic usage of Iterator in the Dacapo pmd benchmark discussed in Section 4.

Another limitation, which is shared by most dynamic mining techniques, is that the inferred property is often specific to the base program used to generate the traces. For example, the usage pattern of Iterator mined in our experiment is createIter (hasNext next)* hasNext?. But the most general pattern allowed by the Java API specification, although uncommon in practice, is createIter (hasNext+ next)* hasNext?, which allows multiple hasNext before a next. To overcome this limitation, we use at least two different base programs for each set of target classes. But, similar to the above argument, the generated application-specific property is actually useful for detecting possible behavioral errors.

The third limitation is related to choosing a proper event set $\mathcal{E}$. Including irrelevant events in $\mathcal{E}$ may bring noise into the mining process, resulting in unnecessarily complicated automata and hard-to-understand regular patterns. Even when all the events in $\mathcal{E}$ are relevant, some of them may play the same role in the specification and can be grouped to achieve more compact results. We applied the following heuristics to minimize the impact of this limitation in our experiments. First, the methods of a target class are grouped according to common prefixes or suffixes, such as get and set. Methods in the same group generates the same events. Second, for classes containing more method groups than a given threshold, 5 in our experiments, we ignore *pure methods*, i.e., methods that do not change the target object, according to the name of the method group, e.g., get and is. Third, after the automaton is inferred, events that cause the same transitions are grouped using disjunctions and reported by our tool. One can choose to name the grouped events to achieve a more compact specification. For example, the original pattern mined for the Collection-Iterator example in Figure 1 was (add || remove || clear)* (createIter(hasNext next)* hasNext? (add || remove || clear)*)?. We renamed (add || remove || clear) to update to achieve the more concise result in Figure 1. Figure 4 was modified accordingly. The experiments show that these heuristics can greatly improve quality of resulting specifications without requiring much effort on the user.

4 Evaluation

We have implemented our mining approach in a tool prototype for Java, named jMiner, and applied it to a set of programs. In our experiments, we manually wrote ApsectJ programs to log uses of target classes. After the logged traces were collected, they were sliced for different combinations of target classes, which were manually chosen according to method declarations of involved classes. We believe this process can be achieved automatically but it is not focus of this paper. Table 2 summaries our experiments. All the experiments were carried out on an a machine with 1.5GB RAM and a Pentium 4 2.66GHz processor. The operating system used was Ubuntu Linux 7.10. The first two columns of Table 2 tell those packages and classes on which we applied our tool to infer specifications. Our experiments covered different kinds of applications for diversity to achieve a comprehensive evaluation of our approach. They include Java base libraries, Apache Lucene [18], an open source text search engine library, jFreeChart [12], a library for displaying charts in Java applications, and apache JAMES [11], an open source mail server. The third column gives the base programs used in our experiments. We used the Dacapo benchmark suite [3] in most experiments, SCAN [23],

applications	classes	base programs	trace lengths	numbers of trace slices	analysis time (second)			mined properties
					slicing	learning	refinement	
Java IO library	Reader, Writer	Dacapo	9000	1143∼2285	0.9	0.01	0.1	4
Java Util library	Collection, Map, Iterator	Dacapo	50000	79∼12855	1.3	0.07	0.2	5
Apache Lucene	Document, IndexWriter	Dacapo, SCAN	29000	792∼2025	1.0	0.04	0.1	3
jFreeChart	Plot,jFreeChart,Listener	Dacapo, jFreeChart tests	8900	14∼14	0.8	0.01	0.2	6
Apache JAMES	CommandHandler, SMTPHandler	JAMES test cases	300	11∼29	0.01	0.01	0.01	2

Table 2. Summary of experiments on jMiner

a desktop content search engine using Lucene, and existing tests for jFreeChart and JAMES. At least two base programs are executed for every set of target classes [3].

The fourth column gives the average length of logged traces. It turned out that the length of the logged trace (the number of events) is not directly related to quality of the mined property. For example, the shortest trace, i.e., the one for JAMES, resulted in a quite comprehensive and precise specification of the SMTP protocol, as discussed below. The fifth column shows the numbers of non-parametric trace slices computed from the logged parametric traces. The numbers of slices are given as ranges because the number of trace slices is equivalent to the number of parameter instances and is different from one parameter set to another. For example, in the Util experiment, 12855 Iterator instances were observed, while we only saw 79 Map objects. Column 6 to 8 give the analysis time for every stage in our approach, showing that our algorithm is efficient. The last column in Table 2 gives numbers of properties mined in our experiments which describe meaningful system behaviors.

4.1 Problem Revealed

We manually compared specifications generated from different base programs if they appeared to be different. Our inspection revealed three problematic behaviors of the Dacapo benchmark. The first is caused by the misuse of Iterator in the pmd benchmark mentioned in Section 1. Our tool mined the following pattern, createIter next || createIter (hasNext next)* hasNext?, from the benchmark, indicating that the program tried to fetch (and only fetched) the first element in an iterator without checking hasNext. The examination of the implementation confirmed our observation and showed that this is a potential bug in the program, which has been fixed in a newer version of pmd.

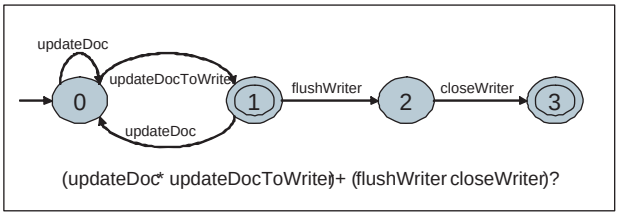

Figure 11. Inferred problematic DocumentIndexWriter **pattern**

Another suspicious behavior was known to us ([6]), related to the uses of Writer in the Xalan benchmark. Our tool mined the following pattern createWriter (write* closeWriter)$^+$ from the benchmark, immediately showing that the program may write a Writer even after it has been closed. The third problem is related to the uses of the Lucene library in Dacapo's luindex benchmark, as explained in what follows. In our experiments, we chose two main classes, namely, Document and IndexWriter, from Lucene to infer possible specifications. Document is the unit of indexing and search and IndexWriter creates and maintains an index. Usually, to create indexing information, a Document instance is created to represent the indexing information for a file, then it is added to a IndexWriter object that will write the indexing information to the index database later. We used two base program for Lucene in our experiments, namely, the luindex benchmark in Dacapo and SCAN. Figure 11 gives an inferred specification involving both Document and IndexWriter, in which updateDoc means updates on the document, addDocToWriter means adding the document to an indexwriter, and flushWriter and closeWriter are events to flush and close the indexwriter, respectively.

The inspection of the inferred specification raised two unexpected behaviors to us. First, since state 1 is the final state, it indicates that some executions did not flush the writer at the end, which can also be easily found from the corresponding regular pattern. A quick check showed that the inferred specification from SCAN had only state 3 as the final state and the problem was clearly caused by the luindex benchmark. A further inspection showed that the benchmark may only create and modify indexing information but not flush them before exit. In some cases, the garbage collector will close the writer and flush out the information, but in other cases, such behavior will lead to loss of information in the writer that is not flushed promptly.

The other unexpected behavior captured in the specification is the transition from state 1 to state 0, which enforces the Document object to be added into the IndexWriter again after some changes are made. A search in the Lucnene's documentation shows that it is indeed a required pattern of using IndexWriter: the changes on the Document object are *not visible* to the indexwriter if they are made after the document is added to the indexwriter.

4.2 More Results

We next discuss some inferred specifications in depth.

jFreeChart: Chart and Plot. jFreeChart [12] is a Java chart library that makes it easy for developers to display

[3]Dacapo is regarded as a suite of base programs.

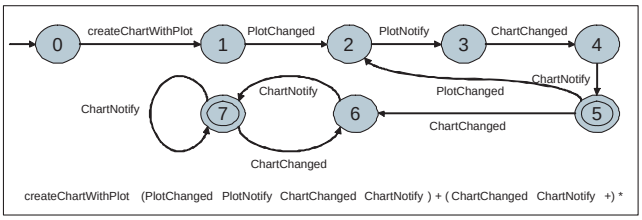

Figure 12. Chart-Plot **pattern in jFreeChart**

professional quality charts in their applications. Our evaluation used three core classes/interfaces in the jFreeChart package, namely, jFreeChart, the main entry of the package, Plot that draws charts and Listener that is the interface used for the listener pattern. In this package, both the jFreeChart and Plot classes provide a large number of functions, resulting in complicated specifications if we try to cover all of them. Also, such comprehensive specifications are likely to be meaningless since many functions are not relevant to one another. Therefore, in addition to grouping the functions according to their prefixes and suffixes, we limited our experiments to take at most two groups of functions for each class into account every time when inferring a possible specification. Figure 12 shows the specification involving the Changed and Notify function groups in both jFreeChart and Plot. The Changed functions are functions used to notify the jFreeChart/Plot object that some related components have been changed. The Notify functions are used by the jFreeChart/Plot object to notify other related components. Also, CreateChartWithPlot is the creation of a jFreeChart using a Plot object. Although the inferred automaton seems non-trivial, it is straightforward to understand the behavior from the generated regular pattern: if a chart is created using a plot then every time the plot is changed, the chart will be notified and changed accordingly but not the other way around. In addition, every time the plot or the chart is changed, it will also send out notifications. But a chart may send out notifications that are not caused by the Changed function (from the ChartChanged ChartNotify+ pattern). Actually, our experiments also inferred other specifications, showing that other functions, such as Set, may also lead to notifications in a chart.

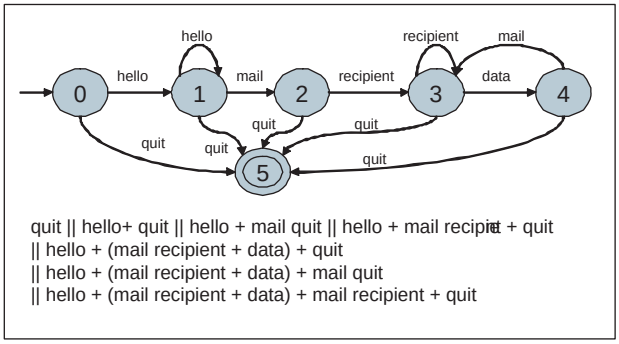

Figure 13. Mail server protocol in JAMES

Apache JAMES: CommandHandler. We applied our tool to the CommandHanler interface and the SMTPHandler class in JAMES' SMTPServer package. A few meaningful specifications were mined but we focus on the specification related to CommandHandler here. CommandHandler defines the core interface to handle commands from the mail client. Its mined specification, as showed in Figure 13, precisely describes the SMTP protocol without using authentication (due to the lack of test cases). hello is the command to initiate a mailing session, mail sets the sender of an email, recipient adds recipients of the email, data gives the content of the email, and quit aborts the session, which can occur at any point. It is worth noting that, unlike other examples where the method name is used as the event id and the target object and arguments are used as event parameters, we used the class name of the target CommandHandler object, e.g., HeloCmdHandler, as the event id and the argument of the onCommand() function in the CommandHandler interface as the event parameter, which is an SMTPSession object representing a mailing session. It is easy to achieve such changes in our approach not only because AspectJ provides the needed programming capability but also because the parametric slicing algorithm and the mining algorithm in our approach do not depend on any specific meaning of the event and the trace. This way, our approach provides the user the flexibility to apply domain knowledge in specification mining to achieve better results.

5 Related Work

Many approaches have been proposed to mine automata-based specifications from observed execution traces. [2] introduces a mining tool, Strauss, to mine automata-based specifications from execution traces. Strauss uses flow analysis and type inference to decide the relation among parameters carried by events, mainly because Strauss is applied to C programs that often use primitive values to represent different entities in the systems. We focus on mining properties about objects in this paper and can relate parameters by simply checking whether they refer to the same object at the same position. But we believe that our parametric trace slicing algorithm is general and can be applied to parameters of primitive types once combined with the analysis proposed in [2]. Strauss is also based on the PFSA algorithm, but it requires the user to manually choose "hot spots" in order to improve the resulting automaton. Our learning algorithm automatically refines the automaton using the input trace set. [13] is another approach using the PFSA algorithm to infer typestate specifications. But it makes no effort to improve the inferred automaton, leaving necessary correction of the specification to the user.

[25] and [9] present techniques to mine fixed types of patterns from execution traces, e.g., two-letter alternating patterns ((a b)*) or three-letter patterns. Comparing with our approach, they are not able to mine complex specifica-

tions but they do not require any pre-defined symbols to use in the specification. [1] proposes an approach to efficiently infer state-based specifications using frequent closed partial orders (FCPO). But it cannot handle more complicated patterns like in Figure 1. [17] presents an advanced algorithm to mine extended finite state machines (EFSM), finite state machines extended with state constraints. Conceptually, EFSM subsumes the parametric specification discussed in this paper since a parameter binding can be regarded as a state constraint. However, such generalization of EFSM requires having different sets of states for different parameter bindings in the generated automaton. As we show in the experiments, the number of parameter bindings can be large in practice and may result in a very complicated automaton using EFSM. On the other hand, it can be beneficial to combine EFSM with our parametric trace slicing framework to handling both parameters and state constraints.

Some static analysis based approaches have also been proposed to infer automata-based specifications, including [20], [21] and [24]. Static mining approaches avoid the limited coverage of dynamic mining but they are usually less scalable and create more redundancy in the results. Also, there are many dynamic analysis approaches to infer specifications other than finite state machines. For example, [4] and [16] mine sequence diagram-based specifications, and Daikon [7] and [10] infer state predicates.

6 Conclusion

This paper presented an approach to mine parametric state-based specifications from observed execution traces. The approach is based on a general framework that generates non-parametric traces from the logged parametric traces, thus allowing one to apply mining techniques that do not take parameters into account. A PFSA-based algorithm to infer compact and precise state-based specifications from non-parametric traces was developed within the parametric framework. The approach has been implemented in a tool prototype, named jMiner. Our experiments show that our technique is effective in mining specifications that involve one or more parameters for different types of programs. The results are usually compact, precise and easy to understand, facilitating human inspection and program analysis. Inspection of the inferred specifications revealed problematic behaviors in some of the analyzed programs.

References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *FSE'07*, 2007.

[2] G. Ammons, R. Bodik, and J. R. Larus. Mining specification. In *PLDI'02*, 2002.

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

[4] L. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of uml sequence diagrams for distributed java software. *IEEE Trans. Software Engineering*, 32(9):642–663, 2006.

[5] J. A. Brzozowski. Derivatives of regular expressions. *Journal of ACM*, 11(4):481–494, 1964.

[6] F. Chen and G. Roşu. MOP: An efficient and generic runtime verification framework. In *OOPSLA*, 2007.

[7] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE'00*, 2000.

[8] J. Feldman and A. Biermann. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, C-21(6):592–596, 1972.

[9] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *ICSE'08*, 2008.

[10] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE'02*, 2002.

[11] Apache james project. http://james.apache.org/.

[12] jfreechart website. http://www.jfree.org/jfreechart.

[13] P. Joshi and K. Sen. Predictive typestate checking of multithreaded java programs. In *ASE'08*, 2008.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, 2001.

[15] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97*, 1997.

[16] D. Lo and S. Maoz. Mining scenario-based triggers and effects. In *ASE'08*, 2008.

[17] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE'08*, 2008.

[18] Apache lucene project. http://lucene.apache.org/.

[19] A. V. Raman and J. D. Patrick. The sk-strings method for inferring pfsa. In *ICML'97*, 1997.

[20] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *ICSE'07*, 2007.

[21] M. K. Ramanathan, A. Grama, and S. Jagannathan. specification inference using predicate mining. In *PLDI'07*, 2007.

[22] G. Roşu and F. Chen. Parametric Trace Slicing and Monitoring. Technical Report UIUCDCS-R-2008-2977, University of Illinois at Urbana-Champaign, 2008.

[23] Scan website. http://scan.sourceforge.net/.

[24] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *ISSTA'07*, 2007.

[25] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal api rules from imperfect traces. In *ICSE'06*, 2006.