Mining Parametric Specifications

Choonghwan Lee Feng Chen Grigore Roşu
Department of Computer Science
University of Illinois at Urbana-Champaign
{clee83, _, grosu}@illinois.edu

ABSTRACT

Specifications carrying formal parameters that are bound to concrete data at runtime can effectively and elegantly capture multi-object behaviors or protocols. Unfortunately, parametric specifications are not easy to formulate by non-experts and, consequently, are rarely available. This paper presents a general approach for mining parametric specifications from program executions, based on a strict separation of concerns: (1) a trace slicer first extracts sets of independent interactions from parametric execution traces; and (2) the resulting non-parametric trace slices are then passed to any conventional non-parametric property learner. The presented technique has been implemented in jMiner, which has been used to automatically mine many meaningful and non-trivial parametric properties of OpenJDK 6.

1. INTRODUCTION

Formal specifications define behaviors that systems or parts of systems should obey. A parametric specification is a formal specification that carries parameters that are bound to concrete object instances at runtime. As an example, Figure 1 shows a finite state automaton (FSA) that describes a parametric specification involving two Java classes, Collection and Iterator. Each edge represents an event, such as calling a method. ⟨init⟩ represents calling the constructor of Collection and update calling a method that changes the contents of the Collection object specified as a parameter c; add, remove and clear can be such methods. iterator represents creating an Iterator object for a Collection object and has two parameters: the underlying Collection object and the created Iterator object. hasNext and next represent invocations on methods hasNext and next of Iterator, respectively. The next method returns the next element in the iteration and the hasNext method checks if the iteration has more elements (i.e., hasNext checks if next is safe). The specification in Figure 1 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; indeed, once the FSA enters state 4, no method calls of the iterator are allowed (state 4 does not accept hasNext or next). A violation of this property results in a runtime exception. Figure 1 also shows a typical usage pattern of Iterator: every call to next is usually guarded by a call to hasNext.

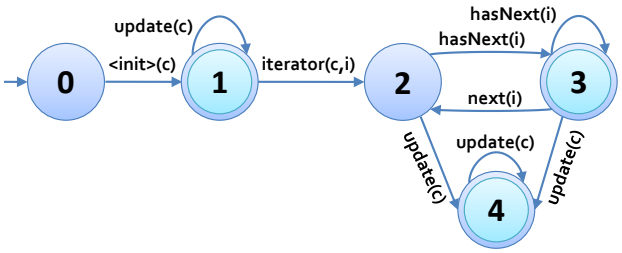

Figure 1: Collection-Iterator **protocol mined with jMiner.**

The use of parameters is crucial in order to properly distinguish among different object interactions. Indeed, if one omits the parameters of the specification in Figure 1, then one would wrongly report a protocol violation when, e.g., two consecutive next calls are observed on two distinct iterators. Current parametric specification monitoring approaches maintain a clean separation between different object interactions, each interaction being observed by a distinct monitor; if an event is relevant to several monitors, e.g., an update event of a collection c which is relevant to all iterators i over c, then that event is dispatched to all interested monitors [5]. This way, monitors are not perturbed by irrelevant events or by unfortunate interaction interleavings. The major objective of this paper is to present a technique that achieves the same degree of separation between different object interactions, but for mining instead of monitoring.

Numerous specification mining approaches have been proposed, e.g., [2, 16, 22, 10, 9, 1, 15, 18, 19, 20, 8, 3, 13, 14, 23, 7] among others. Parametric specifications are much more challenging to mine than non-parametric ones, mainly due to the complexity of handling parameters. Indeed, as discussed in Section 7, none of the existing approaches provides a satisfactory solution for mining parametric specifications in their full generality and invulnerability to "unfortunate" interaction interleaving. For example, *none of the approaches* that we are aware of is able to mine the specification in Figure 1 in the presence of arbitrary interaction interleavings.

Contributions. In this paper we present an effective and generic dynamic approach for mining parametric specifications. We strongly separate the tasks of parameter handling and of specification learning, which has two major benefits: (1) parameter handling makes no assumption on the types of specifications to mine, resulting in a generic parametric mining framework; and (2) learning is not affected by parameters or perturbed by interaction interleavings, reducing the

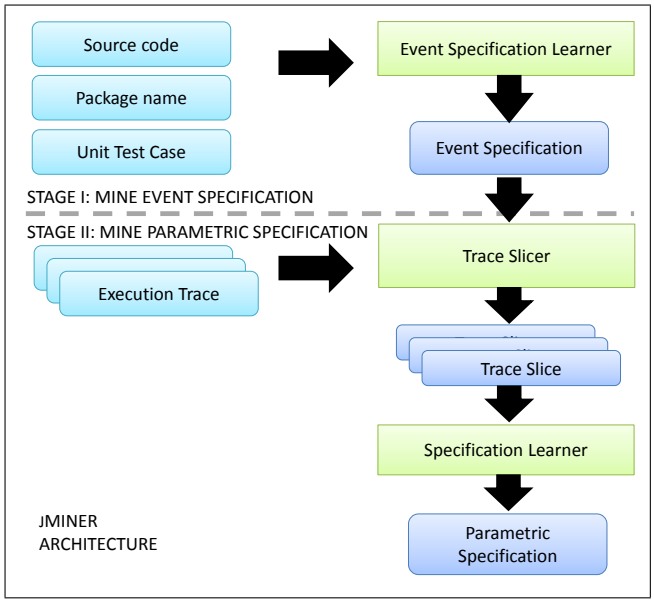

Figure 2: Mining Parametric Specifications.

effort to re-use existing algorithms (or to develop new ones) and increasing the overall precision of mining. The presented approach has been implemented in and extensively evaluated with JMINER. The major tasks performed by JMINER are: instrument the observed program to emit parametric events of interest at runtime; run the instrumented program and collect a parametric execution trace; slice the parametric trace into a set of independent object interactions, called trace slices; pass the resulting trace slices to any learner; finally, put everything together into one or more parametric specifications. When used on packages that provide unit tests, JMINER provides a *completely automated* parametric specification mining solution: building upon the hypothesis that unit tests are devised to stress interactions among objects and methods that are likely to obey some protocol, JMINER learns the events of interest by first running and observing the existing unit tests. This worked well in practice and, indeed, JMINER has been used to automatically mine many properties of OpenJDK6, like the one in Figure 1.

The rest of this paper is organized as follows. Section 2 highlights our overall approach to mining parametric specifications. Section 3 explains our novel trace slicing technique. Section 4 describes the other parts of our framework in detail. Section 5 discusses experiments and limitations. Section 7 discusses related work. Section 8 concludes.

2. APPROACH OVERVIEW

Here we give a high-level overview of our mining approach.

DEFINITION 1. *(**Event specification**) We write **methods** as $m(T_t, T_r, T_{p_1}, \ldots, T_{p_n})$, where m is the method name, T_t is the receiver type, T_r is the return type, and $T_{p_1}, \ldots, T_{p_n}$ are the types of its parameters; for uniformity, we call each of $T_t, T_r, T_{p_1}, \ldots, T_{p_n}$ a **method parameter**. If M is a set of methods, let X_M be all the method parameters of reference type for all methods in M. An **event specification** is a pair $\langle M, X \rangle$, where M is a set of methods and $X \subset X_M$.*

For example, consider a set of methods $M = \{$Iterator.hasNext (Iterator, boolean), Iterator.next(Iterator, Object), Collection.iterator (Collection, Iterator)$\}$. Then, $X_M = \{$Collection, Iterator, Object$\}$, because boolean is not a reference type. $\langle M, X_M \rangle$ forms an

```
1.  ArrayList.add(ArrayList:158)
2.  AbstractList.iterator(ArrayList:158, AbstractList$Itr:119)
3.  AbstractList$Itr.hasNext(AbstractList$Itr:119)
4.  AbstractList$Itr.next(AbstractList$Itr:119)
5.  AbstractList$Itr.hasNext(AbstractList$Itr:119)
6.  AbstractList.iterator(ArrayList:263, AbstractList$Itr:131)
7.  AbstractList$Itr.hasNext(AbstractList$Itr:131)
8.  AbstractList$Itr.next(AbstractList$Itr:131)
9.  AbstractList$Itr.next(AbstractList$Itr:119)
```

Figure 3: Fragment of an execution trace.

event specification, but one may prefer event specification $\langle M, X \rangle$, where X drops Object from X_M. An event specification defines a set of related methods and their parameters that would likely form a useful parametric specification.

Parametric specifications are specifications adding parameters to properties specified using any (non-parametric) trace formalism [4, 5]. Here we only consider regular patterns, definable through FSA; we only used FSA learners so far.

DEFINITION 2. *A **parametric specification** is a tuple $(S, M, X, s_0, \delta, F)$, where $\langle M, X \rangle$ is an event specification and (S, M, s_0, δ, F) is an FSA, where S is the set of states, $s_0 \in S$ is the initial state, $\delta : [S \times M \rightharpoonup S]$ is the transition (partial) function, and $F \subseteq S$ is the set of final states.*

Figure 1 shows one such parametric specification. As a notational convenience, we write methods in M with their parameters in X (and drop the other parameters in $X_M - X$).

Our parametric specification mining approach consists of two stages, also giving the architecture of JMINER, as depicted in Figure 2: event specification learning (Section 4.1) and parametric specification mining (Sections 3 and 4.2). The former yields a set of event specifications. The latter mines a parametric specification for each event specification.

Providing precise event specifications is inconvenient, since it requires expert knowledge about the target software. A set of overly diverse methods would result in a too complex specification likely to be application-specific, while a set of too few methods would result in a too simple specification covering only part of the usage pattern. Our approach is to automatically learn event specifications from unit tests, when available, based on the hypothesis that unit tests check the behavior of tightly interacting objects and thus the methods involved in such interactions likely obey some specification.

The parametric specification mining stage takes an event specification and various program execution traces as input, and yields a parametric specification as output. To obtain program execution traces, we wrote a Java Virtual Machine Tool Interface (JVMTI)[11] agent that logs information about the invoked methods and their arguments, and attached it to the JVM. JVMTI provides a convenient means to access the call stack and to uniquely identify objects. Our JVMTI agent records all method invocations from all threads in chronological order, so that interactions that span over multiple threads can be recognized. The events and parameters which are not relevant for the given event specification are then filtered out. Figure 3 shows a resulting execution trace fragment when the event specification is $\langle$\{Collection.add, Collection.iterator, Iterator.hasNext, Iterator.next\}, \{Collection,Iterator\}$\rangle$. Each line corresponds to a method invocation and contains a method name together with the paired actual type and object identifier for each method parameter. Note that our execution trace filtering takes into account Java's implicit upcasting. For example, ArrayList.add is not filtered out because the method add is defined by ArrayList's superclass Collection and is included in the event specification.

As explained in Section 1, the event parameters in ex-

ecution traces play a crucial role. Indeed, if one dropped the parameters then one would see just one large and likely unreal (and thus unsuitable for mining) interaction that interleaves many different object interactions. For example, in the trace in Figure 3, events 8 and 9 are unrelated since their Iterator objects are different. However, a specification learner making abstraction of parameters would infer a faulty specification allowing two consecutive nexts.

To infer accurate specifications no matter how interactions interleave, independent interactions need to be considered separately. To this end, our approach introduces *trace slicing*. Informally, an interaction is a series of events that manipulate a set of objects. For example, two interactions exist in Figure 3: [add, iterator, hasNext, next, hasNext, next] (events 1, 2, 3, 4, 5 and 9) on ⟨ArrayList:158, Abstract$List:119⟩ and [iterator, hasNext, next] (events 6, 7 and 8) on ⟨ArrayList:263, Abstract$List:131⟩. Our trace slicer identifies such interactions and passes them separately to the specification learner.

A specification learner finally infers a parametric specification from the set of separate interactions. The learner need not be aware of parameters and, consequently, one can reuse existing learners and/or easily develop new ones. If an FSA learner is employed, it would infer the pattern of alternation between hasNext and next from the above two separate interactions. The inferred FSA is finally annotated with parameters, which produces a parametric specification.

3. SLICING TRACES

In this section, we first define *trace slicing*, a process that dispatches events (in the given execution trace) to trace slices corresponding to different parameter bindings, according to the given event specification. We then introduce the concept of *complete* and *connected* parameter instances in order to remove trace slices that are meaningless and thus can generate noise in the process of mining specifications. We show that a parametric trace can comprise, in the worst-case scenario, exponentially many trace slices (corresponding to complete and connected parameter instances).

3.1 Parametric Trace Slicing

Our terminology used in this section is borrowed from [5].

DEFINITION 3. *Let $\mathcal{E}$ be a set of non-parametric events, called* ***base events*** *or simply* ***events****. An* ***$\mathcal{E}$-trace****, or* ***trace****, is a finite sequence of events in $\mathcal{E}$, i.e., an element in $\mathcal{E}^*$. We write $e \in w$ when event $e \in \mathcal{E}$ appears in trace $w \in \mathcal{E}^*$.*

Let $[A \to B]$ and $[A \rightharpoonup B]$ 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 4. ***(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 (Def. 3), 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 $[X \rightharpoonup V_X]$. Let $\mathsf{Dom}(\theta)$ be $\{x \in X \mid \theta(x)$ defined$\}$ and $\bot \in [X \rightharpoonup V_X]$ be the map undefined everywhere; i.e., $\mathsf{Dom}(\bot) = \emptyset$. Partial maps in $[X \rightharpoonup V_X]$ may also be called* ***parameter instances*** *or* ***parameter bindings****. A* ***parametric trace*** *is a trace with events in $\mathcal{E}\langle X\rangle$, that is, a word in $\mathcal{E}\langle X\rangle^*$.*

In Figure 3, $\mathcal{E} = \{$add, iterator, hasNext, next$\}$, $X = \{$Collection, Iterator$\}$ and $V_X = \{158, 119, \ldots\}$. In the parametric trace shown in Figure 3, add and ⟨ArrayList:158⟩ are the base event and resp. the parameter binding of the first parametric event.

DEFINITION 5. *We say that θ' is* ***less informative*** *than θ, written $\theta' \sqsubseteq \theta$, iff for any $x \in X$, if $\theta'(x)$ is defined then $\theta(x)$ is also defined and $\theta'(x) = \theta(x)$.*

For example, ⟨ArrayList:158⟩ $\sqsubseteq$ ⟨ArrayList:158, Abstract$List:119⟩.

DEFINITION 6. ***(Trace slicing)*** *Given parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$ and partial function θ in X, let the* ***θ-trace slice*** *$\tau{\restriction_\theta}$ of τ be the non-parametric trace in $\mathcal{E}^*$ defined as follows:*

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

A trace slice $\tau{\restriction_\theta}$ first filters out all the parametric events that are irrelevant to the parameter instance θ. For example, when the given parameter instance is ⟨ArrayList:158 Abstract$List:119⟩ in Figure 3, the resulting trace slice does not contain the sixth event. Similarly, events 7 and 8 are also filtered out. Event 1, which binds only ArrayList, is added to the trace slice corresponding to ⟨ArrayList:158, Abstract$List:119⟩ because ⟨ArrayList:158⟩ $\sqsubseteq$ ⟨ArrayList:158, Abstract$List:119⟩.

A trace slice $\tau{\restriction_\theta}$ also forgets the parameter bindings of parametric events. As a result, a trace slice is non-parametric and merely a list of base events. For example, the trace slice corresponding to ⟨ArrayList:158, Abstract$List:119⟩ is [add, iterator, hasNext, next, hasNext, next]. Dropping parameter information enables the parametric specification mining stage to use any learners as long as they take as input a set of strings, where a string is a list of base events.

Although the intuition is clear, developing efficient and correct trace slicing algorithms is non-trivial. First, traversing the trace more than once is undesirable due to efficiency concerns. Second, an event may contain an incomplete binding of the given set of parameters. For example, for the trace in Figure 3, if we choose {Collection, Iterator} as the set of parameters, an add event contains only a Collection parameter, leaving the Iterator parameter unbound. Furthermore, an event may belong to multiple trace slices because its parameter instance can be less informative than many other parameter instances introduced by the trace. For example, if the trace in Figure 3 contained another event iterator(ArrayList:158,AbstractList$List:219), the first event would also belong to the trace slice of ⟨ArrayList:158, Abstract$List:219⟩.

3.2 Complete and Connected Instances

Since a trace slice corresponds to a parameter instance, selecting an appropriate set of parameter instances is crucial in order to prevent meaningless trace slices (such trace slices may result in learning inaccurate specifications). To select only appropriate parameter instances, we use two criteria: completeness and connectedness.

A parameter instance is *complete* if $\mathsf{Dom}(\theta) = X$, where X is the set of parameters in the given event specification. Incomplete parameter instances are considered inappropriate, and trace slices corresponding to those parameter instances are suppressed; i.e., $\tau{\restriction_\theta}$ is suppressed if $\mathsf{Dom}(\theta) \neq X$. For example, if $X = \{$Collection, Iterator$\}$ in Figure 3, the trace slice corresponding to ⟨Abstract$List:119⟩ is suppressed. The trace slice for this incomplete parameter instance is indeed meaningless because it does not represent an interaction between a collection and an iterator.

For some cases, no event provides a complete parameter instance. One such example is illustrated in Figure 4, when $X = \{$Socket, SocketInputStream, SocketOutputStream$\}$. The first

```
1. Socket.<init>(Socket:262)
2. Socket.getInputStream(Socket:262, SocketInputStream:227)
3. Socket.getOutputStream(Socket:262, SocketOutputStream:288)
4. SocketInputStream.read(SocketInputStream:227)
5. Socket.getOutputStream(Socket:562, SocketOutputStream:588)
```

Figure 4: Part of an execution trace used for mining the specification in Figure 10.

four events are part of one interaction ⟨Socket:262, SocketInputStream:227, SocketOutputStream:288⟩, but there is no event that provides the complete parameter instance. Parameter instances from multiple events therefore need to be combined.

DEFINITION 7. *Two parameter instances θ and θ' are **compatible** iff for any $x \in Dom(\theta) \cap Dom(\theta')$, $\theta(x) = \theta'(x)$. We can **combine** compatible instances θ and θ', written $\theta \sqcup \theta'$:*

$$(\theta \sqcup \theta')(x) = \begin{cases} \theta(x) & \text{when } \theta(x) \text{ is defined} \\ \theta'(x) & \text{when } \theta'(x) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

For example, ⟨Socket:262, SocketInputStream:227⟩ is compatible with ⟨Socket:262, SocketOutputStream:288⟩ but is not compatible with ⟨Socket:562, SocketOutputStream:588⟩. Two parameter instances disagreeing on any parameter are incompatible, and thus cannot be combined. ⟨Socket:262, SocketInputStream:227⟩ ⊔ ⟨Socket:262, SocketOutputStream:288⟩ is ⟨Socket:262, SocketInputStream:227, SocketOutputStream:288⟩, obtaining a complete instance when $X = \{$Socket,SocketInputStream,SocketOutputStream$\}$.

Combining multiple parameter instances is therefore necessary for achieving a complete parameter instance. However, if done blindly, it may introduce spurious parameter instances. For example, ⟨Socket:562, SocketInputStream:227, SocketOutputStream:588⟩, obtained by combining ⟨Socket:562, SocketOutputStream:588⟩ and ⟨SocketInputStream:227⟩ in Figure 4, is spurious because an interaction involving all three objects does not exist in the given trace. Trace slices corresponding to spurious parameter instances represent non-existing interactions, and consequently, become noise. To filter out spurious parameter instances, we introduce the concept of *connected* parameter instances.

DEFINITION 8. *If $\tau \in \mathcal{E}\langle X\rangle^*$, we define τ-**connectedness** of parameter instances θ as follows: 1) if $e\langle\theta\rangle \in \tau$ then θ is τ-connected; and 2) if θ_1, θ_2 are τ-connected, compatible, and $\theta_1 \sqcap \theta_2 \neq \bot$, then $\theta_1 \sqcup \theta_2$ is also τ-connected.*

Therefore, a parameter instance is τ-connected iff it is formed by combining parameter instances of events in τ that share parameter bindings. For example, ⟨Socket:262, SocketInputStream:227, SocketOutputStream:288⟩ is τ-connected in Figure 4 because of events 2 and 3, but ⟨Socket:562, SocketInputStream:227, SocketOutputStream:588⟩ is not. In cases where there is no ambiguity, we will say *connected* instead of τ-connected.

Connectedness is motivated by the following observation: in most cases we are interested in mining specifications for a set of *interacting objects*; if two objects appear in the same event, then they interact with each other. Therefore, all the objects contained in a connected parameter instance directly or indirectly interact with one another. Experiments using our technique, discussed in Section 5, show that passing only the trace slices corresponding to connected parameter instances to the specification learner (and discarding the other trace slices) effectively removes noise in the mining process, resulting in accurate specifications.

Computing all possible connected parameter instances is hard. One should not mistakenly think that this problem reduces to computing the ordinary connected components

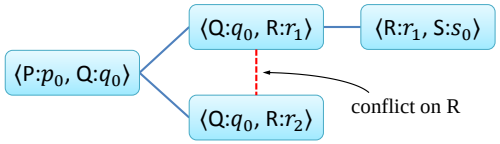

Figure 5: A graph showing that connectedness in a graph does not indicate τ-connectedness.

of a graph, where a vertex represents a parameter instance and an edge exists iff the two associated parameter instances (θ_1 and θ_2) are compatible and $\theta_1 \sqcap \theta_2 \neq \bot$. Figure 5 shows one such graph. P, Q, R and S are parameters, and p_0, q_0, r_1, r_2 and s_0 are parameter values. The graph-connected component in Figure 5 correctly suggests that ⟨P:p_0, Q:q_0⟩ ⊔ ⟨Q:q_0, R:r_1⟩ ⊔ ⟨R:r_1, S:s_0⟩ is τ-connected. However, it also suggests that ⟨P:p_0, Q:q_0⟩ ⊔ ⟨Q:q_0, R:r_2⟩ ⊔ ⟨R:r_1, S:s_0⟩ is τ-connected, which is wrong. Indeed, computing the graph-connected components does not take into consideration the compatibility between parameter instances, while computing the τ-connected parameter instances must. For example, ⟨Q:q_0, R:r_1⟩ and ⟨Q:q_0, R:r_2⟩ are incompatible, but the standard graph-connected component fails to recognize it.

3.3 Complexity of Trace Slicing

In what follows we calculate the worst-case complexity of the trace slicing problem in terms of the number of trace slices as a function of the total length n of the original parametric trace and the size of X, the set of parameters. More precisely, we show that there are approximately[1] $(\frac{n}{m})^m$ trace slices in the worst case when $m \geq 1$, where $m+1$ is the size of X. Note that if $|X| = 1$ then we have at most n trace slices and they are easy to compute. However, if $|X| = \frac{n}{2} + 1$ then we have $2^{\frac{n}{2}}$ trace slices, showing that the addition of conflicting edges (like in Figure 5) makes the graph-connected component problem harder. The maximum of $(\frac{n}{m})^m$ is actually reached when $m = \frac{n}{e}$, in which case it becomes $e^{\frac{n}{e}}$.

Suppose that $X = \{$P$_0$, P$_1$, …, P$_m\}$ for some $m > 0$ and that $\tau = e_1\langle\theta_1\rangle\, e_2\langle\theta_2\rangle \ldots e_n\langle\theta_n\rangle$. The worst case is when any two events have at least one common parameter value, so that $\theta \sqcap \theta' \neq \bot$ for any two parameter instances θ and θ' such that $e\langle\theta\rangle, e'\langle\theta'\rangle \in \tau$; we can achieve that with minimal resources, by designating a parameter instance ⟨P$_0$:p_0⟩ and assuming that that is common to all events. Each event may be in conflict with a certain number of other events. For example, suppose that $e_1\langle\theta_1\rangle$ is in conflict with $a_1 - 1$ events on parameter P$_1$, where $a_1 > 0$. The other $a_1 - 1$ events are also in conflict with each other, so we have a "cluster" of a_1 events which are in conflict with each other on parameter P$_1$. The worst case is when the conflicting a_1 events are in conflict with no other event and when, for each trace slice corresponding to the remaining events, each of them yields a new trace slice. Thus, assuming that the remaining events generate s trace slices, we have $a_1 \times s$ slices in total. We can iterate the argument above and obtain $a_1 \times a_2 \times \cdots \times a_m$ trace slices when we split the n events of τ into clusters of $a_1, a_2, \ldots, a_m$ events with $a_1 + a_2 + \cdots + a_m = n$, each cluster containing those events conflicting on precisely one of the parameters P$_1$, P$_2$, …, P$_m$, respectively. Note that this is not only an over-approximation. It can actually happen, as shown in Figure 6. The product is maximized when $a_1 =$

[1]We do an approximate analysis, making abstraction of the fact that m may not divide n, etc.

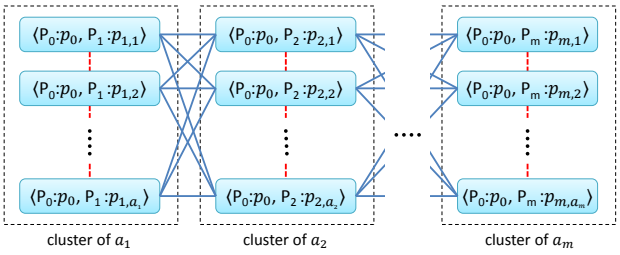

Figure 6: Clusters of $a_1, a_2, \ldots, a_m$ events.

$a_2 = \cdots = a_m = \frac{n}{m}$, in which case it becomes $(\frac{n}{m})^m$.

Therefore, assuming that X is fixed a priori, as it is usually the case, we can only have a polynomial (in the length of the original parametric trace) number of trace slices. If X is not fixed, then one can actually fabricate an absolute worst-case scenario, which maximizes $(\frac{n}{m})^m$. This case occurs when $m = \frac{n}{e}$, in which case the number of trace slices is exponential: $e^{\frac{n}{e}}$. Although it is little likely in practice that the size of X is co-related to the length of the trace, it is instructive to have a clear understanding of the worst-case complexity of the problem that we are attempting to solve.

3.4 Slicing Algorithm

As discussed in Section 3.3, the number of trace slices is $(\frac{n}{m})^m$ in the worst case. Since all trace slices can be distinct, this number gives a lower bound for all trace slicing algorithms. This lower bound is hard to achieve, though, since computing complete and connected parameter instances may require several operations of instance combination. For example, $\langle \mathsf{P}_0{:}p_0,\ \mathsf{P}_1{:}p_{1,1},\ \mathsf{P}_2{:}p_{2,1},\ \ldots,\ \mathsf{P}_m{:}\ p_{m,1}\rangle$ in Figure 6 can be obtained only after at least m instance combination operations: $((\langle \mathsf{P}_0{:}p_0,\ \mathsf{P}_1{:}p_{1,1}\rangle \sqcup \langle \mathsf{P}_0{:}p_0,\mathsf{P}_2{:}p_{2,1}\rangle) \sqcup \langle \mathsf{P}_0{:}p_0,\mathsf{P}_3{:}p_{3,1}\rangle) \sqcup \ldots \sqcup \langle \mathsf{P}_0{:}p_0,\ \mathsf{P}_m{:}p_{m,1}\rangle)$. Furthermore, a trace slicing algorithm needs to search for compatible parameter instances in order to create combined parameter instances.

Our trace slicing algorithm in Figure 7, called Slicer, traverses the given trace only once and avoids the construction of meaningless trace slices. Slicer has two stages: (1) it first processes the entire parametric trace, event by event, constructing intermediate results Δ; and (2) then it constructs the set of trace slices Ψ, each corresponding to a complete and connected parameter instance.

During the first stage, Slicer stores in Δ intermediate trace slices only for parameter instances that actually occur in observed events. Neither combined parameter instances nor trace slices for them are created at this stage. The second stage, ConstructConnected, constructs Ω holding all possible connected parameter instances by combining compatible parameter instances in the loop at lines 2–3. For each complete and connected parameter instance, its corresponding trace slice is finally constructed at lines 4–6. Γ collects all intermediate trace slices corresponding to θ's sub-instances. MergeTraces is essentially the merge function of merge sort, using the position of events in the trace for comparison (events in trace slices are listed chronologically).

Theorem 1. *After running* Slicer *on* $\tau \in \mathcal{E}\langle X\rangle^*$

1. *$\Psi(\theta)$ is defined iff θ is τ-connected and* $\mathsf{Dom}(\theta) = X$.
2. *If $\Psi(\theta)$ is defined, then $\Psi(\theta) = \tau{\restriction_\theta}$.*

This theorem tells all trace slices can be retrieved from Ψ.

We next analyze the complexity of Slicer. It first calls HandleEvent n times, and, assuming that a self-balancing

Input : X, $\tau = e_1\langle\theta_1\rangle\, e_2\langle\theta_2\rangle\, \ldots\, e_n\langle\theta_n\rangle$
Output: $\Psi \in [[X \to V_X] \to \mathcal{E}^*]$
Global : $\Delta \in [[X \to V_X] \to \mathcal{E}^*]$

Function Slice()
1 **for** $i \leftarrow 1$ **to** n **do** HandleEvent($e_i\langle\theta_i\rangle$)
2 ConstructConnected()

Function HandleEvent($e\langle\theta\rangle$)
1 **if** $\Delta(\theta)$ *undefined* **then** $\Delta(\theta) \leftarrow \epsilon$
2 $\Delta(\theta) \leftarrow \Delta(\theta)\, e$

Function ConstructConnected()
1 $\Omega \leftarrow \{\theta \mid \Delta(\theta)$ is defined$\}$
2 **while** $\exists\theta_1, \theta_2 \in \Omega$ *compatible*, $\theta_1 \sqcap \theta_2 \neq \bot$, $\theta_1 \sqcup \theta_2 \notin \Omega$ **do**
3 $\quad\Omega \leftarrow \Omega \cup \{\theta_1 \sqcup \theta_2\}$
4 **foreach** $\theta \in \Omega$ *s.t.* $\mathsf{Dom}(\theta) = X$ **do**
5 $\quad\Gamma = \{\Delta(\theta') \mid \theta' \sqsubseteq \theta$ and $\Delta(\theta')$ is defined$\}$
6 $\quad\Psi(\theta) \leftarrow$ MergeTraces(Γ)

Figure 7: Slicer: Trace Slicing algorithm.

binary search tree is used for Δ, the complexity of HandleEvent is $O(\log n)$. The loop at lines 2–3 in ConstructConnected can pick θ_1 and θ_2 from $\Omega \times \Omega$, and each iteration takes $O(m)$ time for checking the compatibility and combining the two parameter instances. There are $|\Omega|$ iterations of the loop at lines 4–6, with each iteration taking $O(m)$ time. The running time of the entire algorithm is thus $O(n \log n + |\Omega|^2 \cdot m + |\Omega| \cdot m) = O(n \log n + |\Omega|^2 \cdot m)$. Since the algorithm creates all possible connected parameter instances, $|\Omega|$ can be calculated as follows: the number of connected parameters with $|\mathsf{Dom}(\theta)| = i + 1$ is $\binom{m}{i} \cdot (\frac{n}{m})^i$ because we can choose i parameters and there are $\frac{n}{m}$ parameter values for each parameter. Thus, we have $|\Omega| = \sum_{i=1}^{m} \binom{m}{i} \cdot (\frac{n}{m})^i = (\frac{n}{m} + 1)^m$, and the time complexity of Slicer is $O(n \log n + (\frac{n}{m} + 1)^{2m} \cdot m) = O((\frac{n}{m} + 1)^{2m} \cdot m)$. As for the space complexity, it needs to maintain $O(|\Omega|)$ connected parameter instances of length $O(m)$ during trace slicing. It also needs space for $(\frac{n}{m})^m$ trace slices of size m as illustrated in Figure 6. Therefore, the space complexity is $O((\frac{n}{m} + 1)^m \cdot m + (\frac{n}{m})^m \cdot m) = O((\frac{n}{m} + 1)^m \cdot m)$.

Slicer iterates through all possible connected parameter instances in the loop at lines 2–3 in ConstructConnected. In our experiments, this step was comparatively the most expensive wrt performance, so we have investigated several possibilities to optimize it. We next describe two of our optimizations which bring considerable performance benefits. Instead of blindly picking a pair of parameter instances from Ω and combining them, our implementation proceeds in a bottom-up manner. At the first step, it picks two parameter instances (θ_1 and θ_2) such that $|\mathsf{Dom}(\theta_1)| = |\mathsf{Dom}(\theta_2)| = N$, and creates $\theta_1 \sqcup \theta_2$, if necessary. After handling all parameter instances with N parameter bindings, it picks parameter instances with $N+1$ parameter bindings, and so on and so forth until N reaches the size of X, the set of parameters. This way, a parameter instance is considered for compatibility within only a limited window, reducing the number of iterations.

Our second optimization is to group parameter instances so that all parameter instances in the same group bind exactly the same parameters. Grouping also reduces the number of iterations at lines 2–3 in ConstructConnected. For example, if $\langle \mathsf{P}{:}p_1, \mathsf{Q}{:}q_1\rangle$ is chosen as θ_1, all parameter instances that belong to the group corresponding to $\{\mathsf{R}, \mathsf{S}\}$ will be excluded from the list of candidates for θ_2 because any parameter instance in this group would result in $\theta_1 \sqcap \theta_2 = \bot$.

4. LEARNING IN JMINER

Here we discuss how the trace slicing technique in Section 3 is incorporated within our JMINER parametric mining approach, by means of discussing JMINER's event specification learner (from unit tests) as well as its specification learner (based on a refinement of the off-the-shelf PFSA).

4.1 Learning Event Specifications

The event specification learner dynamically infers a set of event specifications from the target software. It takes as input the source code, a target package name, and unit tests (all for the target software). Providing these inputs is easy and requires no expert knowledge. For example, in order to mine specifications in the java.util package of OpenJDK 6, what the user needs to provide is: the source code of OpenJDK 6, the target package name java.util, and the unit tests for java.util. Here is, e.g., a typical unit test in OpenJDK 6:

```
1.  import java.util.*;
2.  public class CheckForComodification {
3.    private static final int LENGTH = 10;
4.    public static void main(String[] args) throws Exception {
5.      List<Integer> list = new ArrayList<Integer>();
6.      for (int i = 0; i < LENGTH; i++)  list.add(i);
7.      try {  for (int i : list)
8.                if (i == LENGTH - 2)  list.remove(i);
9.      } catch(ConcurrentModificationException e) { return; }
10.     throw new RuntimeException
11.             ("No ConcurrentModificationException");
12.   } }
```

This unit test case is written to test if a concurrent modification of a Collection object is detected and a runtime exception is raised. As the Java compiler translates the for- each loop at lines 7–8 into ArrayList.iterator, AbstractList$Itr.hasNext and AbstractList$Itr.next, an execution of this test case will reveal an interaction between ArrayList and AbstractList$Itr.

An advantage of using unit tests for learning event specifications is that tightly interacting objects are well isolated. For example, the unit test case above considers only one issue, namely detecting a concurrent modification. This isolation avoids tangential relationships among issues, which are usually application-specific and thus not likely to obey a generic specification. For example, an interaction on a FileReader object and an interaction on a FileWriter object can be related in an application through a File object, because both FileReader and FileWriter objects can be constructed with the same File object. Although somewhat related, it is expected that reading and writing are two separate issues and thus specifications involving both are unnecessary. Such tangential relationships rarely occur in unit tests. Moreover, users can obtain unit tests for free as many software packages are shipped with them. Unit tests are well maintained as they are frequently run and failed cases are promptly addressed.

Learning related methods and parameters. We first trace unit test executions using a JVMTI agent like the one in Section 2, but one which also logs the thread identifier and the depth of the call stack in front of each event (each thread has its stack and every method invocation is logged). Here is, e.g., part of the trace logged by the unit test above:

```
1. 1 2 ArrayList.<init>(ArrayList:689)
2. 1 2 ArrayList.add(ArrayList:689, Integer:830)
3. 1 3 ArrayList.ensureCapacity(ArrayList:689)
4. 1 2 AbstractList.iterator(ArrayList:689, AbstractList$Itr:950)
5. 1 2 AbstractList$Itr.hasNext(AbstractList$Itr:950)
6. 1 3 ArrayList.size(ArrayList:689)
7. 1 2 AbstractList$Itr.next(AbstractList$Itr:950, Integer:821)
```

One can infer that add invoked ensureCapacity; and ⟨init⟩, add, iterator, hasNext and next were invoked in order by the same method (not shown here), since they have the same thread identifier and depth of the call stack. ⟨init⟩ is a special name reserved for constructors. Although an execution trace from a unit test case usually contains relatively few events, it may still be too large to precisely infer related methods.

We next analyze the execution trace in order to remove irrelevant events. Many heuristics are possible here; we prefer to use two heuristics which were also used in [21, 16] and appear to work well: keep only events corresponding to methods (1) which are defined in the user-specified target package, and (2) which are directly invoked by methods of the class that declares the main entry of a test case. Our rationale for using these heuristics is that unit tests are rarely written for interactions among multiple packages, and that a unit test consists of one core class for performing the actual test and other utility classes for supporting the core class.

All the relevant events are then partitioned into groups of related events. Two events are *directly related* iff they share at least one common argument, and *related* iff they are connected through a sequence of directly related events. For example, ⟨init⟩ (event 1) and add (event 2) are directly related due to ArrayList:689, and ⟨init⟩ (event 1) and next (event 7) are related through iterator (event 4).

An event specification is then created for each partition. For each object used as a receiver in a partition, its type is generalized to the least specific type that specifies all methods involving that object. Then, the least specific type and all the involved methods (after their declaring types are generalized to the least specific types) are added as a parameter and, respectively, methods. For the trace above, e.g., a partition including AbstractList:689 and AbstractList$Itr:950 is first generated. Then, AbstractList:689 is generalized to AbstractList because AbstractList specifies all involved methods. As a result, a parameter AbstractList and the generalized methods (AbstractList.⟨init⟩, AbstractList.add and AbstractList.iterator) are added. Similarly, AbstractList$Itr:950 adds a parameter Iterator and two methods (Iterator.hasNext and Iterator.next). These two parameters and five methods form an event specification.

Filtering out generics parameters. Generics may yield undesirable event specifications. Consider, for example, the following execution trace (Locale is a generic type):

```
1.  ArrayList.<init>(ArrayList:62)
2.  ArrayList.add(ArrayList:62, Locale:57)
3.  AbstractList.iterator(ArrayList:62, AbstractList$Itr:519)
4.  AbstractList$Itr.hasNext(AbstractList$Itr:519)
5.  AbstractList$Itr.next(AbstractList$Itr:519, Locale:57)
6.  ArrayList.<init>(ArrayList:1279)
7.  ArrayList.add(ArrayList:1279, Locale:57)
```

The above event specification learner would identify [hasNext, next, ⟨init⟩, add] (events 4, 5, 6 and 7) as the interaction on ⟨ArrayList:1279, AbstractList$Itr:519, Locale:57⟩ (ArrayList and AbstractList$Itr are subclasses of AbstractList and Iterator, respectively). This interaction is spurious as it is about conceptually unrelated objects ArrayList:1279 and AbstractList$Itr:519 (AbstractList$Itr:519 is an iterator for ArrayList:62), which happen to be related due to Locale:57.

The above spurious interaction is caused by the fact that AbstractList.add and Iterator.hasNext bring Locale to the event specification. A general way to prevent this problem is to recognize parameters of generic types and avoid adding them to the event specification. The rationale is that the instantiated types were unknown at the time the generic class was written and, consequently, they should be kept separate.

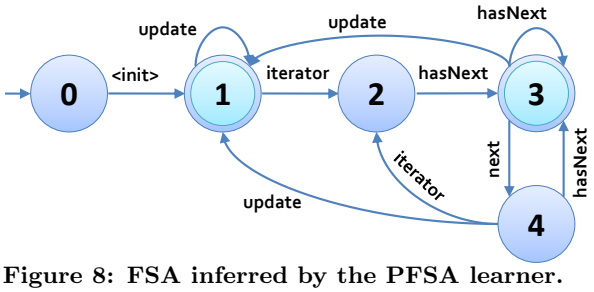

Figure 8: FSA inferred by the PFSA learner.

To detect parameters of generic types, JMiner reads the generic signature (which tells the parameters of generic type) for each method using JVMTI. Above, it detects that the parameter of AbstractList.add and the return type of Iterator.next are generic, and thus it does not add Locale to the event specification. Then Locale:57 can be ignored and the trace slicer will not recognize the spurious interaction.

4.2 Learning Parametric Specifications

JMiner takes the event specifications learned from unit tests as explained above and passes them to its core component, the trace slicer. As discussed in Sections 3 and 2, the trace slicer takes an event specification and an execution trace produced by a program that exercises the target package, and produces a set of trace slices, each corresponding to an observed interaction. The generated trace slices are then passed to any non-parametric specification learner. We included one such specification learner in JMiner, which we discuss below. Our learner consists of two components: an off-the-shelf PFSA learner and an FSA refiner.

PFSA learner. A PFSA is an FSA where each transition is labelled with how often the transition occurs. A PFSA learner takes a set of strings as input and infers a PFSA. Several PFSA learning approaches have been proposed [17]; we here adopt the sk-strings algorithm [17], since it performs well at inferring small FSAs. It first constructs an FSA that precisely accepts the input set of strings. Each transition is then annotated with a frequency, saying how many times that transition was observed. The sk-strings algorithm then generalizes by merging states which are *sk-equivalent*: two states are sk-equivalent iff corresponding sets of bounded strings (ones that are frequently generated from each of the two states) are matched. As a result of this approximation, two states can be merged even when they are not strictly equivalent, making it possible for the inferred FSA to accept not only the input strings but also other "similar" strings. The reader is referred to [17] for more details. After running the sk-strings algorithm, the learner drops the frequency information, yielding an ordinary FSA. Figure 8 shows the FSA inferred by the sk-strings algorithm for our CollectionIterator example (Figure 1 shows the desired specification).

FSA refiner. Although PFSA learner's approximations are generally desirable in many application domains, the resulting FSAs turned out to often be overly general in our domain of mining specifications from execution traces, in that the mined FSAs accept a large number of undesirable traces. For example, the trace [⟨init⟩, iterator, hasNext, next, iterator, hasNext] is accepted by the FSA in Figure 8, but it is impossible to occur in any program execution (only one iterator event can be observed in any interaction between a collection and an iterator). To prevent over-generalized specifications, our specification learner implemented in JMiner refines the FSA inferred by the sk-strings algorithm.

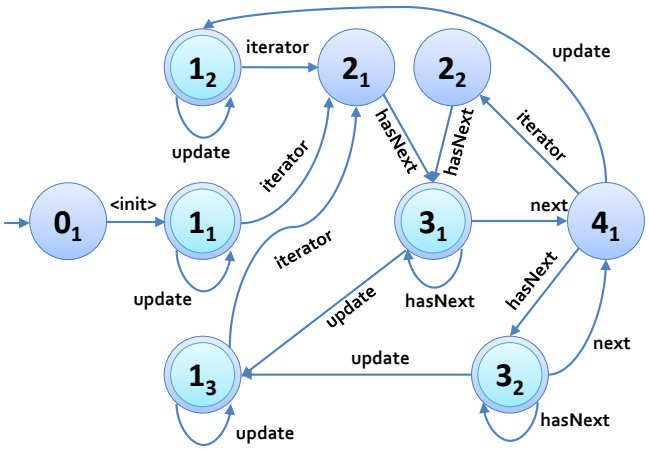

Figure 9: Expanded FSA of Figure 8.

The overall goal of our refiner is to eliminate transitions caused by over-generalization while keeping desirably generalized transitions. An obvious step for avoiding over generalization is to remove all the transitions that are never taken by any of the trace slices provided as the input of the specification learner. For example, the iterator transition from state 4 to state 2 in Figure 8 is never taken (the same Iterator object cannot be created twice), so it can be safely removed. However, that is not enough: the resulting FSA still accepts infeasible interactions containing multiple iterators, e.g., [⟨init⟩, iterator, hasNext, update, iterator, hasNext].

The fundamental problem is that PFSA learners do not take the context into account when merging states, while the context is important in programming languages. For example, the two transitions from states 3 and 4 to state 1 in Figure 8 were obtained by the PFSA learner by merging two contextually different states. Modifying a PFSA learner to take the context into account seems hard. Instead, our approach is to "reverse engineer" the FSA generated by the PSFA learner to partly take the context into account. More precisely, our refiner first expands the inferred FSA as follows: if a state s has n incoming edges from the other states, then s is replaced by n corresponding states $(s_1, s_2, \ldots, s_n)$ in order to differentiate the originating state. Figure 9 shows the expanded FSA of the one in Figure 8. States 1_2 and 1_3 indicate that an iterator has been already used, whereas state 1_1 indicates that no iterators have been used.

Using the expanded FSA, the refiner removes all the transitions that are never taken by the given trace slices (i.e., caused by over-generalization), in our case $1_2 \to 2_1$, $1_3 \to 2_1$, and $4_1 \to 2_2$. Then the refiner finally eliminates the states with no incoming transitions and merges states that have the same outgoing transitions. For example, state 2_2 is eliminated, and states $1_2, 1_3$ and states $3_1, 3_2$ are merged, respectively. The resulting FSA for the one in Figure 9 is shown in Figure 1. The reader is referred to [6] for more details.

5. EVALUATION OF JMINER

We applied JMiner to mining specifications in four packages of OpenJDK 6: java.util, java.io, java.lang and java.net. OpenJDK 6 contains various unit tests and is well documented, allowing us to validate the mined specifications. Selecting unit tests of a specific package (for event specification learning) is easy because the unit tests are well structured.

We used execution traces obtained from the DaCapo bench-

Target package	Event Specifications		Parametric Specifications	
	# files	# events	# files	# events
java.util	370	65,854,349	14	88,999,435
java.io	382	28,835,588		
java.lang	372	41,784,568		
java.net	221	9,429,744	31	10,938,168

Table 1: Traces used in the experiments.

Target package	Event Specifications	Parametric Specifications	
		Slicing	Learning
java.io	24	115	24
java.lang	38	112	75
java.net	59	14	1
java.util	59	133	86

Table 2: Times (minutes; Windows, 3GHz, 1GB).

mark suite 9.12 and the Apache JAMES Server 2.3.1 as training sets for specification mining. DaCapo contains 14 programs and provides a harness to execute each program and validate an execution. We used DaCapo for mining specifications in java.util, java.io and java.lang, and Apache JAMES (which contains several test cases for whole-system checking) for mining specifications in java.net. Table 1 shows information on the traces used in our experiments. Two kinds of traces were used: traces for learning event specifications and traces for mining parametric specifications. We limited the execution time for each program in DaCapo to one hour.

The event specification learning stage automatically inferred 51,271 event specifications. jMiner has an aggressive filter which automatically discards most of the learned event specifications (though the user can manually intervene and save or adjust any event specification). The filter first deletes methods that can be called anytime (such as getters returning primitive types, toString(), and hashCode()) and thus are likely not involved in protocols, and then removes all event specifications having only one non-constructor method (these lead to obvious patterns). Only 498 different event specifications passed jMiner's filter. 230 of them resulted in parametric specifications, but the other 268 did not, because DaCapo and Apache JAMES generated no interactions corresponding to them:

Target package	# event specifications	# parametric specifications
java.io	145	66
java.lang	82	48
java.net	90	36
java.util	181	80

Table 2 shows the elapsed execution time for three separate stages: event specification learning (Section 4.1), trace slicing (Section 3), and specification learning (Section 4.2). Table 2 does not include the time spent on running unit tests and applications. Each number represents the total elapsed time; e.g., learning 145 event specifications for java.io took 24 minutes. Trace slicing accounted for most of the time. These traces contain an enormous number of events and parameter instances, and our slicer relates events whenever they share the same objects, no matter how far these events are from each other in the trace. Overall, considering that high-quality specifications are invaluable, we believe that the time spent on trace slicing is a minor aspect.

We next discuss some parametric specifications that were automatically mined by jMiner. More can be found at [12].

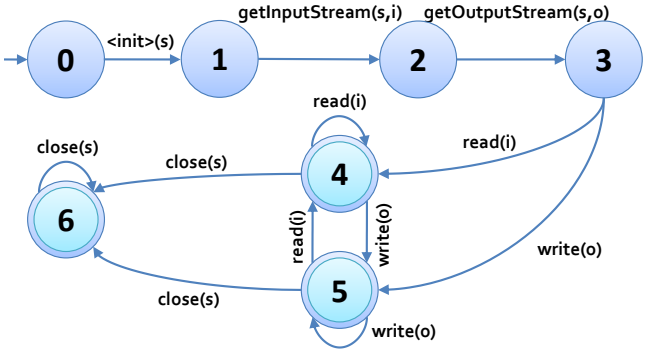

Figure 10: Socket **specification mined using jMiner**

Client socket. Figure 10 shows a specification of a client-side stream socket. The constructor of Socket connects a new socket to the peer specified by its arguments. Then, getInputStream and getOutputStream return the input stream and the output stream, respectively, which enable data transmission using read and write. The specification states that data transmission can be repeatedly performed in arbitrary order until the socket is closed, which is consistent with the documentation. It also states that close can be invoked multiple times, which is undocumented but correct. The specification also correctly suggests that the invocation of close is optional because states 4 and 5 are also final states. In fact, calling close is recommended, but not mandatory because the connection is eventually closed when the Socket object is reclaimed.

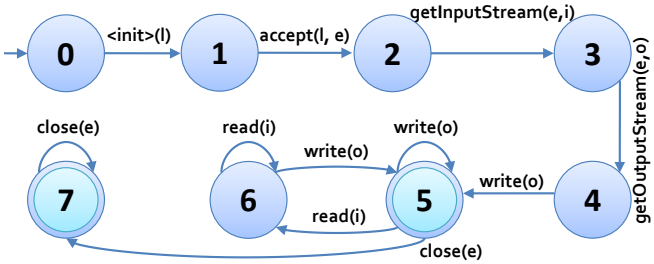

Figure 11: ServerSocket **specification mined with jMiner.**

ServerSocket. Figure 11 shows an inferred specification for the server-side socket. After a ServerSocket object l is instantiated, accept listens for a connection and accepts it, returning a new socket e. getInputStream and getOutputStream return an InputStream object i and an OutputStream object o respectively, which can be used for data transmission. After these operations, close can be invoked to close the connection. This behavior spans over multiple threads in most cases because multiple clients can connect to the same port represented by a single ServerSocket object, and a server needs to handle them simultaneously. The trace slices in our experiments indeed involved two threads: the data transfer was processed in a separate thread. If each thread's trace was considered separately like in [16], the specification could not be mined.

Collection, iterator. Figure 1 shows a specification of Collection and Iterator. As discussed in Section 1, the specification states a safety property of Collection and a typical usage pattern of Iterator. In order to mine the specification in Figure 1, we slightly modified the automatically inferred event specification that defines the five methods Collection.⟨init⟩, Collection.add, Collection.iterator, hasNext and next. Knowing that add, remove and clear are similar (all these methods update the collection), we grouped the three methods into one hypothetical method, which we called update.

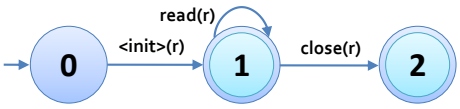

Figure 12: Reader **specification mined using jMiner.**

Reader, writer. Figure 12 shows a specification of a Reader object, stating that read can be repeatedly called before close. It does not enforce the invocation of close, similarly to the Socket specification above. JMINER also mined a similar specification for Writer. These specifications are simple, but can detect the wrong invocation of read or write after close.

6. LIMITATIONS

We have identified, during our experiments, a few limitations of our approach. First, the learning process is limited to the observed behaviors. This is an inherent limitation of all dynamic approaches. For example, the specifications in Figures 10 and 11 wrongly enforce the order between getInputStream and getOutputStream because this was consistently observed in the training set. Also, in Figure 13 one may expect that the specification should state that nextToken is guarded by hasMoreTokens. Surprisingly, the inferred one allows the invocation of the two methods in an arbitrary order since it was actually observed that Xalan, an application in DaCapo, calls nextToken without calling hasMoreTokens.

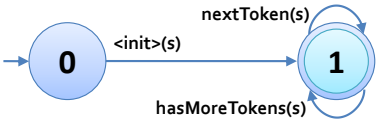

Figure 13: StringTokenizer **spec. mined using jMiner.**

After inspecting the source code of Xalan, we could see that the pattern is not defective because Xalan first retrieves the number of tokens by calling countTokens and then consecutively calls nextToken as many times as specified by countTokens. Due to countTokens, a specification on StringTokenizer cannot be stricter than Figure 13. Considering countTokens as well does not improve the specification because our technique cannot infer that the return value of this method indicates the number of allowed nextToken calls. This limitation is inherent to all FSA-based approaches: FSA cannot count.

7. RELATED WORK

Ammons et al. [2] mines specifications from execution traces and user-provided input: functions of interest, attributes for those functions, and a scenario seed. Their tool extracts a set of API usage scenarios from execution traces and then passes it to a PFSA learner. Providing attributes requires in-depth knowledge (one should understand the side-effect of each function); the user should imagine a hypothetical object corresponding to a scenario, and should mark a parameter as *define* or, respectively, as *use* if the parameter changes or depends upon the state of the object. Scenarios are identified starting from the seed event, searching the execution trace along *define-use* chain. Having explicit seed events and using the chain reduce the search space, but it may result in failing to recognize complete interactions. For example, assume the following trace:

```
1. ArrayList.add(Collection:1)
2. AbstractList.iterator(Collection:1, Iterator:2)
3. AbstractList$Itr.hasNext(Iterator:2)
4. AbstractList.iterator(Collection:1, Iterator:3)
5. AbstractList$Itr.hasNext(Iterator:3)
6. ArrayList.add(Collection:1)
```

Also suppose that the seed event is iterator (it is the only event that connects Collection and Iterator) and that add *defines* Collection, iterator *defines* Iterator and *uses* Collection, and hasNext *uses* Iterator. For example, event 2 depends on event 1 as event 1 *defines* Collection:1 and event 2 *uses* Collection:1. From these inputs, two scenarios will be extracted: 1 2 3 and 1 4 5. None of them are complete with regards to the interaction between Collection and Iterator: none includes event 6 because add does not *use* Collection and, consequently, cannot be reached from iterator in a scenario. Since the retrieved scenarios are incomplete, the PFSA learner will eventually infer a restricted FSA as shown in Figure 14. Stating that add *defines* Collection and also *uses* it does not solve the problem. In fact, it causes another problem: a scenario that includes operations on two distinct iterators will be extracted.

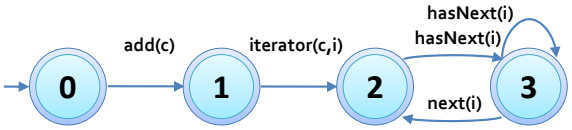

Figure 14: Restricted Collection–Iterator **usage pattern.**

Pradel and Gross [16] propose a dynamic mining technique based on collecting from execution traces a list of related receiver-method pairs up to a user-specified level of nested method calls. Unlike ours, their technique does not consider individual interactions separately. Therefore, it may merge individual interactions and thus infer inaccurate specifications. For example, if the execution trace in Figure 3 is observed within a method, their technique will not consider the two interactions separately and, consequently, infer a faulty specification that allows consecutive calls to next. Moreover, it cannot infer a specification that spans over multiple threads, since it creates a separate trace for each thread; e.g., the specification in Figure 11 cannot be mined. Furthermore, it may fail to mine specifications from distantly related events if the value of the level of nested method calls is too small. If, on the other hand, the value is too large, it may produce specifications that include too many methods and would likely be application-specific.

Yang et al. [22] propose a technique to find all pairs of methods that satisfy the predefined particular pattern $(ab)^*$ from execution traces. Although their chaining heuristic composes somewhat more complex patterns, such as $(abc)^*$ (by connecting related specifications into a chain), it cannot infer useful and more complex patterns like in Figure 1. Gabel and Su [9] extend [22]; their work considers an additional pre-defined pattern $(ab^*c)^*$. It then combines instances of these basic patterns, generating complex patterns. Unlike our approach, it neglects parameters; thus, it may infer noisy specifications from sequences of irrelevant events that happen to match the predefined patterns.

Dallmeier et al. [7] also present a technique for mining FSAs from execution traces. A state in the FSA inferred by their work represents the results of *inspector* methods that observe the internal state, such as isEmpty and hasNext, whereas a state in our approach is abstract (e.g., "before using an iterator"). Associating each state with inspectors

can help users to easily understand the specification, but it fails to capture implicit states such as "an iterator for a collection is being used" because no methods in Collection can observe it. In our approach, the sequence of method calls can capture those states. Moreover, their work considers only one object and is essentially non-parametric.

Lorenzoli et al. [15] gives an advanced algorithm to mine extended finite state machines (EFSMs), i.e., FSMs extended with state constraints. Their approach proposes no trace slicing technique, which is at the core of our approach; instead, they assume that the training traces are already given as separate interactions with constraints. In principle, one could use state constraints to encode parameter instances, but considering the huge number of parameter instances encountered in our experiments, we believe that would be impractical. We have only looked at mining parametric FSAs; it may, however, be beneficial to plug their learner into jMiner and use the latter for learning EFSMs from trace slices produced by our algorithm in Section 3.

Acharya et al. [1] proposes a static technique that generates a set of traces along possible execution paths directly from the source code, and then produces an API usage pattern from it. Since it mines partial orders, the resulting specifications cannot describe loops; thus, it cannot mine the specifications shown in this paper. Zhong et al. [23] also presents a static mining technique for sequential patterns from open source repositories. Unlike our approach, their tool does not consider individual interactions separately. For example, if there are multiple distinct interactions on Collection in a method, their tool can extract a faulty method call sequence. Since their tool inlines multiple methods, the probability that a method call sequence consists of multiple interactions on Collection is high, which makes this approach improper to mine specifications of frequently used classes.

Chen and Roşu [5] introduce trace slicing for monitoring. There is an inherent duality between parametric specification monitoring and parametric specification mining: they both rely on a parametric trace slicing process for identifying interactions, followed either by monitoring the resulting trace slices against given specifications in the first case, or by inferring the specifications that best explain the observed trace slices in the second case. However, a blind use of off-the-shelf trace slicing techniques for monitoring leads to noisy and inefficient trace slicers for mining. It is allowed for trace slicers for monitoring to generate trace slices corresponding to incomplete or unconnected parameter instances because such trace slices can be ignored by the underlying monitor. In the context of mining, however, such trace slices would result in faulty specifications.

8. CONCLUSION

This paper presented a generic and automated approach to mine accurate parametric specifications from execution traces with minimal effort. The event specification learner reduces the effort by automatically inferring event specifications from unit tests, and the trace slicer identifies independent interactions, allowing one to apply various learning techniques that do not handle parameters. Moreover, our automaton refiner makes the inferred specifications more accurate by eliminating spurious transitions. Our experiments indicate that the technique is effective: it mined many meaningful specifications that involve multiple parameters.

9. 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*, 2007.

[2] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, 2002.

[3] 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.

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

[5] F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *TACAS*, 2009.

[6] F. Chen, C. Lee, and G. Roşu. Mining parametric specifications. Technical report, University of Illinois, 2010. URL `http://hdl.handle.net/2142/15109`.

[7] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *WODA*, 2006.

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

[9] M. Gabel and Z. Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *FSE*, 2008.

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

[11] Java Virtual Machine Tool Interface (JVMTI). `http://java.sun.com/javase/6/docs/technotes/guides/jvmti/index.html`.

[12] jMiner Webpage. http://fsl.cs.uiuc.edu/jMiner.

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

[14] D. Lo and S. Maoz. Mining hierarchical scenario-based specifications. In *ASE*, 2009.

[15] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic gen. of software behavioral models. In *ICSE*, 2008.

[16] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *ASE*, 2009.

[17] A. Raman, J. Patrick, and P. North. The sk-strings method for inferring PFSA. In *ICML*, 1997.

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

[19] M. K. Ramanathan, A. Grama, and S. Jagannathan. Specification inference using predicate mining. In *PLDI*, 2007.

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

[21] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *TACAS*, 2005.

[22] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE*, 2006.

[23] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *ECOOP*, 2009.