# Parametric Trace Slicing and Monitoring*

Feng Chen and Grigore Roşu

Department of Computer Science, University of Illinois at Urbana-Champaign
{fengchen,grosu}@cs.uiuc.edu

**Abstract.** Analysis of execution traces plays a fundamental role in many program analysis approaches. Execution traces are frequently parametric, i.e., they contain events with parameter bindings. Each parametric trace usually consists of many *trace slices* merged together, each slice corresponding to a parameter binding. Several techniques have been proposed to analyze parametric traces, but they have limitations: some in the specification formalism, others in the type of traces they support; moreover, they share common notions, intuitions, even techniques and algorithms, suggesting that a fundamental understanding of parametric trace analysis is needed. This foundational paper gives the first solution to parametric trace analysis that is unrestricted by the type of parametric properties or traces that can be analyzed. First, a general purpose parametric trace slicing technique is discussed, which takes each event in the parametric trace and distributes it to its corresponding trace slices. This parametric trace slicing technique can be used in combination with any conventional, non-parametric trace analysis, by applying the latter on each trace slice. An online monitoring technique is then presented based on the slicing technique, providing a logic-independent solution to runtime verification of parametric properties. The presented monitoring technique has been implemented and extensively evaluated. The results confirm that the generality of the discussed techniques does not come at a performance expense when compared with existing monitoring systems.

## 1 Introduction and Motivation

Parametric traces, i.e., traces containing events with parameter bindings, abound in programming language executions, because they naturally appear whenever abstract parameters (e.g., variable names) are bound to concrete data (e.g., heap objects) at runtime. For example, if one is interested in analyzing collections and iterators in Java, then execution traces of interest may contain events $\mathsf{createIter}\langle c\, i\rangle$ (iterator $i$ is created for collection $c$), $\mathsf{updateColl}\langle c\rangle$ ($c$ is modified), and "$\mathsf{next}\langle i\rangle$ ($i$ is accessed using its next element method), instantiated for particular collection and iterator instances. Most properties of parametric traces are also parametric, i.e., refer to each particular parameter instance; for example, a property may be "collections are not allowed to change while accessed through

* Supported in part by NSF grants CCF-0448501, CNS-0509321 and CNS-0720512, by NASA contract NNL08AA23C, and by several Microsoft gifts.

S. Kowalewski and A. Philippou (Eds.): TACAS 2009, LNCS 5505, pp. 246–261, 2009.

iterators", which is parametric in a collection and an iterator. To distinguish properties parametric in a set of parameters $X$ from ordinary, non-parametric properties, we write them $\Lambda X.P$; for example, violations of the above parametric property expressed as a regular expression (here matches mean violations) can be "$\Lambda c, i.$ createIter$\langle c\, i\rangle$ next$\langle i\rangle^*$ updateColl$\langle c\rangle^+$ next$\langle i\rangle$". From here on we omit the event parameters in parametric properties when they are redundant; for example, we write "$\Lambda c, i.$ createIter next$^*$ updateColl$^+$ next" for the above.

Parametric properties, unfortunately, are very hard to formally verify and validate against real systems, mainly because of their dynamic nature and potentially huge or even unlimited number of parameter bindings. Let us extend the above example: in Java, one may create a collection from a map and use the collection's iterator to operate on the map's elements. A similar safety property is: "maps are not allowed to change while accessed indirectly through iterators". Its violation pattern is: "$\Lambda m, c, i.$ createColl (updateMap | updateColl)$^*$ createIter next$^*$ (updateMap | updateColl)$^+$ next", with two new parametric events createColl$\langle m, c\rangle$ (collection $c$ is created from map $m$) and updateMap$\langle m\rangle$ ($m$ is updated). All the events used in this property provide only partial parameter bindings (createColl binds only $m$ and $c$, etc.), and parameter bindings carried by different events may be combined into larger bindings during the analysis; e.g., createColl$\langle m_1, c_1\rangle$ can be combined with createIter$\langle c_1, i_1\rangle$ into a full binding $\langle m_1, c_1, i_1\rangle$, and also with createIter$\langle c_1, i_2\rangle$ into $\langle m_1, c_1, i_2\rangle$. It is highly challenging for a trace analysis technique to correctly and efficiently maintain, locate and combine trace slices for different parameter bindings, especially when the trace is long and the number of parameter bindings is large.

Parametric properties have been receiving growing interest in *runtime verification* (RV), as shown by the increasing number of RV systems supporting them, e.g., [3,1,14,12,7,13,15,9,5,4]. Most of these techniques tightly couple the handling of parameter bindings with the property checking, yielding monolithic but supposedly efficient monitors. For example, Tracematches [1] extends state machines with parameter bindings in order to support parametric regular pattern properties; a series of optimizations to the resulting data-structures make Tracematches one of the most efficient RV systems [3]. The major challenges these "monolithic monitor" approaches face are how to keep track of the status for each particular parameter instance during property checking, and how to correctly garbage-collect portions of the monitor as they become irrelevant [14,5,4]. Such couplings of parameter binding and property checking result in rather complex and property-formalism-specific algorithms, hard or impossible to adapt to other formalisms. For example, [3] builds upon a *finite* state machine skeleton associated to the underlying pattern, so it cannot be adapted to, e.g., context-free patterns; [14] stacks automata in order to support parametric context-free patterns, making it slower than Tracematches [3] and insensitive to certain events of interest (such as ends of procedures [15]); Eagle [4] has no garbage-collection due to its generality, causing prohibitive overhead [3].

JavaMOP [9] proposes a different solution, based on a complete decoupling of parameter binding from property checking. This separation allows the use

of "off-the-shelf" algorithms and techniques for non-parametric properties as plug-ins; e.g., JavaMOP supports several property specification formalisms, including regular expressions, temporal logics, and context-free patterns [15,9], all parametric. However, the technique currently supported by JavaMOP can only handle a limited type of traces, namely ones in which the first event for a particular property instance binds all the property parameters. This limitation prevents JavaMOP from supporting many useful parametric properties [3]. In this paper we show that the decoupling of parameter binding and property checking is not only possible without any limitation, but also very practical.

In spite of all the recent advances in parametric property and trace analysis, the following questions are still left largely open in their full generality: Given a parametric trace $\tau$ and a parametric property $\Lambda X.P$, what does it mean for $\tau$ to be a good or a bad trace for $\Lambda X.P$? How can we leverage, to the parametric case, knowledge and techniques to analyze conventional, non-parametric traces against conventional, non-parametric properties? In this paper we first formulate and then rigorously answer these questions and empirically validate our answer.

***Contributions.*** Besides proposing a formal semantics to parametric traces, properties, and monitoring, we make two theoretical contributions and discuss an implementation that validates them empirically. Our first result is a general-purpose online parametric trace slicing algorithm (algorithm $\mathbb{A}\langle X\rangle$), which positively answers the question: given a parametric execution trace $\tau$, can one effectively find the slices $\tau \upharpoonright_\theta$ corresponding to each parameter instance $\theta$ without having to traverse the trace for each $\theta$? Our second result, which builds upon the slicing algorithm, is an online monitoring technique (algorithms $\mathbb{B}\langle X\rangle$ and $\mathbb{C}\langle X\rangle$) for parametric properties, which separates handling of parameters from checking trace slices against the specified property. It positively answers the question: is it possible to monitor arbitrary parametric properties $\Lambda X.P$ against parametric execution traces $\tau$, provided that the root non-parametric property $P$ is monitorable using conventional monitors? Finally, we implemented and evaluated the proposed techniques and show empirically that their generality does not come at a performance expense when compared with existing monitoring systems.

***Paper structure.*** Section 2 formalizes parametric events, traces and properties, defines trace slicing and discusses an online trace slicing algorithm. Section 3 presents our main techniques for parametric trace monitoring. Section 4 discusses implementation optimizations to the proposed monitoring technique and its evaluation. Section 5 summarizes related researches and Section 6 concludes.

***Note.*** Proofs to all the claimed results can be found in [16].

## 2 Parametric Trace Slicing for Monitoring

In this section, we first define some basic notions (event, trace and property) and then present an online parametric trace slicing algorithm that provides the foundation for the online monitoring technique discussed in Section 3.

### 2.1 Events, Traces and Properties

Here we introduce the notions of event, trace and property, first non-parametric and then parametric. Trace slicing is then defined as a reduct operation that forgets all the events unrelated to the given parameter instance.

**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*** *when $\mathcal{E}$ is understood or not important, 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 $e \in w$.*

**Example.** *(part 1 of simple running example)* Consider a certain resource (e.g., a synchronization object) that can be acquired and released during the lifetime of a given procedure (between its begin and end). Then $\mathcal{E}$={acquire, release, begin, end} and execution traces corresponding to this resource are sequences of the form "begin acquire acquire release end begin end", "begin acquire acquire", etc. For now there are no "good" or "bad" execution traces. □

**Definition 2.** *An $\mathcal{E}$-**property** $P$, or simply a (base or non-parametric)* ***property****, is a function $P : \mathcal{E}^* \to \mathcal{C}$ partitioning the set of traces into categories $\mathcal{C}$. It is common, but not enforced, that $\mathcal{C}$ includes "validating", "violating", and "don't know" (or "?") categories. In general, $\mathcal{C}$, the co-domain of $P$, can be any set.*

**Example.** *(part 2)* Consider a regular expression specification, (begin($\epsilon$ | (acquire (acquire | release)*release))end)*, stating that the procedure can (non-recursively) take place multiple times and, if the resource is acquired during the procedure then it is released by the end of the procedure. The validating traces are those matching the pattern, e.g., "begin acquire acquire release end begin end". At first sight, one may say that all the other traces are violating traces, because they are not in the language of the regular expression. However, there are two interesting types of such "violating" traces: ones which may still lead to a validating trace provided the right events will be received in the future, e.g., "begin acquire acquire", and ones which have no chance of becoming a validating trace, e.g. "begin acquire release acquire end". Therefore, we can pick $\mathcal{C}$ to be the set {validating, violating, don't know} and, for a given regular expression $E$, define its associated property $P_E : \mathcal{E}^* \to \mathcal{C}$ as follows: $P_E(w) =$ validating iff $w$ is in the language of $E$, $P_E(w) =$ violating iff there is no $w' \in \mathcal{E}^*$ such that $w\,w'$ is in the language of $E$, and $P_E(w) =$ don't know otherwise; this is the monitoring semantics of regular expressions in JavaMOP [9]. Other semantic choices are possible; for example, one may choose $\mathcal{C}$ to be the set {matching, don't care} and define $P_E(w) =$ matching iff $w$ is in the language of $E$, and $P_E(w) =$ don't care otherwise; this is the semantics of regular expressions in Tracematches [1]. □

We next extend the above definitions to the parametric case, i.e., events carrying concrete data instantiating abstract parameters.

**Example.** *(part 3)* Events acquire and release are parametric in the resource; if $r$ is the name of the generic "resource" parameter and $r_1$ and $r_2$ are two concrete resources, then parametric acquire/release events have the form acquire$\langle r \mapsto r_1 \rangle$,

$\mathsf{release}\langle r \mapsto r_2\rangle$, etc. Not all events need carry instances for all parameters; e.g., the begin/end parametric events have the form $\mathsf{begin}\langle\bot\rangle$ and $\mathsf{end}\langle\bot\rangle$, where $\bot$, the partial map undefined everywhere, instantiates no parameter. □

Let $[A \to B]/[A \rightharpoonup B]$ be the sets of total/partial functions from $A$ to $B$.

**Definition 3.** ***(Parametric events and traces).*** *Let $X$ be a set of* ***parameters*** *and let $V$ be a set of corresponding* ***parameter values****. If $\mathcal{E}$ is a set of base events like in Def. 1, then let $\mathcal{E}\langle X\rangle$ be the set of corresponding* ***parametric events*** *$e\langle\theta\rangle$, where $e$ is a base event in $\mathcal{E}$ and $\theta$ is a partial function in $[X \rightharpoonup V]$. A* ***parametric trace*** *is a trace with events in $\mathcal{E}\langle X\rangle$, that is, a word in $\mathcal{E}\langle X\rangle^*$.*

To simplify writing, we occasionally assume the parameter values set $V$ implicit.

**Example.** *(part 4)* A parametric trace can be: $\mathsf{begin}\langle\bot\rangle\ \mathsf{acquire}\langle\theta_1\rangle\ \mathsf{acquire}\langle\theta_2\rangle\ \mathsf{acquire}\langle\theta_1\rangle\ \mathsf{release}\langle\theta_1\rangle\ \mathsf{end}\langle\bot\rangle\ \mathsf{begin}\langle\bot\rangle\ \mathsf{acquire}\langle\theta_2\rangle\ \mathsf{release}\langle\theta_2\rangle\ \mathsf{end}\langle\bot\rangle$, where $\theta_1$ maps $r$ to $r_1$ and $\theta_2$ maps $r$ to $r_2$. We take the freedom to only list the parameter values when writing parameter instances, that is, $\langle r_1\rangle$ instead of $\langle r \mapsto r_1\rangle$, or $\tau\restriction_{r_2}$ instead of $\tau\restriction_{r\mapsto r_2}$, etc. With this notation, the above trace is: $\mathsf{begin}\langle\rangle\ \mathsf{acquire}\langle r_1\rangle\ \mathsf{acquire}\langle r_2\rangle\ \mathsf{acquire}\langle r_1\rangle\ \mathsf{release}\langle r_1\rangle\ \mathsf{end}\langle\rangle\ \mathsf{begin}\langle\rangle\ \mathsf{acquire}\langle r_2\rangle\ \mathsf{release}\langle r_2\rangle\ \mathsf{end}\langle\rangle$. This trace involves two resources, $r_1$ and $r_2$, and really consists of *two trace slices*, one for each resource. The begin and end events belong to both trace slices. The slice corresponding to $\theta_1$ is "begin acquire acquire release end begin end", while the one for $\theta_2$ is "begin acquire end begin acquire release end". □

**Definition 4.** *Partial functions $\theta$ in $[X \rightharpoonup V]$ are called* ***parameter instances****. $\theta, \theta' \in [A \rightharpoonup B]$ are* ***compatible*** *if for any $x \in \mathsf{Dom}(\theta) \cap \mathsf{Dom}(\theta')$, $\theta(x) = \theta'(x)$. We can* ***combine*** *compatible instances $\theta$ and $\theta'$, written $\theta \sqcup \theta'$, as follows:*

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

*$\theta \sqcup \theta'$ is also called the* ***least upper bound (lub)*** *of $\theta$ and $\theta'$. $\theta'$ is* ***less informative*** *than $\theta$, or $\theta$ is* ***more informative*** *than $\theta'$, 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 our example, $\langle\rangle$ is compatible with $\langle r_1\rangle$ and with $\langle r_2\rangle$, but $\langle r_1\rangle$ and $\langle r_2\rangle$ are not compatible; moreover, $\langle\rangle \sqsubseteq \langle r_1\rangle$ and $\langle\rangle \sqsubseteq \langle r_2\rangle$.

**Definition 5.** ***(Trace slicing)*** *Given parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$ and $\theta$ in $[X \rightharpoonup V]$, let the* ***$\theta$-trace slice*** *$\tau\restriction_\theta \in \mathcal{E}^*$ be the non-parametric trace defined as:*

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

The trace slice $\tau\restriction_\theta$ first filters out all the parametric events that are not relevant for the instance $\theta$, i.e., which contain instances of parameters that $\theta$ does not care about, and then, for the remaining events relevant to $\theta$, it forgets the parameters

so that the trace can be checked against base, non-parametric properties. It is crucial to discard parameter instances that are not relevant to $\theta$ during the slicing, including those more informative than $\theta$, in order to achieve a "proper" slice for $\theta$: in our running example, the trace slice for $\langle\rangle$ should contain only begin and end events and no acquire or release. Otherwise, the acquire and release of different resources will interfere with each other in the trace slice for $\langle\rangle$.

One should not confuse extracting/abstracting traces from executions with slicing traces. The former determines the events to include in the trace, as well as parameter instances carried by events, while the latter dispatches each event in the given trace to corresponding trace slices according to the event's parameter instance. Different abstractions may result in different parametric traces from the same execution and thus may lead to different trace slices for the same parameter instance $\theta$. For the (map, collection, iterator) example in Section 1, $X = \{m, c, i\}$ and an execution may generate the following parametric trace: $\mathsf{createColl}\langle m_1, c_1\rangle$ $\mathsf{createIter}\langle c_1, i_1\rangle$ $\mathsf{next}\langle i_1\rangle$ $\mathsf{updateMap}\langle m_1\rangle$. The trace slice for $\langle m_1\rangle$ is updateMap for this parametric trace. Now suppose that we are only interested in operations on maps. Then $X = \{m\}$ and the trace abstracted from the execution generating the above trace is $\mathsf{createColl}\langle m_1\rangle$ $\mathsf{updateMap}\langle m_1\rangle$, in which events and parameter bindings irrelevant to $m$ are removed. Then the trace slice for $\langle m_1\rangle$ is createColl updateMap. In this paper we focus on the trace slicing; more discussion about trace abstraction can be found in [10].

**Definition 6.** *Let $X$ be a set of parameters together with their corresponding parameter values $V$, like in Definition 3, and let $P : \mathcal{E}^* \to \mathcal{C}$ be a non-parametric property like in Definition 2. Then we define the* ***parametric property*** *$\Lambda X.P$ as the property (over traces $\mathcal{E}\langle X\rangle^*$ and categories $[[X \rightharpoonup V] \to \mathcal{C}]$)*

$$\Lambda X.P : \mathcal{E}\langle X\rangle^* \to [[X \rightharpoonup V] \to \mathcal{C}]$$

*defined as $(\Lambda X.P)(\tau)(\theta) = P(\tau\restriction_\theta)$ for any $\tau \in \mathcal{E}\langle X\rangle^*$ and any $\theta \in [X \rightharpoonup V]$. If $X = \{x_1, ..., x_n\}$ we may write $\Lambda x_1, ..., x_n.P$ instead of $(\Lambda\{x_1, ..., x_n\}.P$. Also, if $P_\varphi$ is defined using a pattern or formula $\varphi$ in some particular trace specification formalism, we take the liberty to write $\Lambda X.\varphi$ instead of $\Lambda X.P_\varphi$.*

Parametric properties $\Lambda X.P$ over base properties $P : \mathcal{E}^* \to \mathcal{C}$ are therefore properties taking traces in $\mathcal{E}\langle X\rangle^*$ to categories $[[X \rightharpoonup V] \to \mathcal{C}]$, i.e., to function domains from parameter instances to base property categories. $\Lambda X.P$ is defined as if many instances of $P$ are observed at the same time on the parametric trace, one property instance for each parameter instance, each property instance concerned with its events only, dropping the unrelated ones.

### 2.2 Algorithm for Online Parametric Trace Slicing

Definition 5 illustrates a way to slice a parametric trace for *given* parameter bindings. However, it is not suitable for online trace slicing, where the trace is observed incrementally and no future knowledge is available, because we cannot know all possible parameter instances $\theta$ apriori. We next define an algorithm

$\mathbb{A}\langle X\rangle$ in Fig. 1 that takes a parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$ incrementally, and builds a partial function $\mathbb{T} \in [[X \rightarrow V] \rightarrow \mathcal{E}^*]$ of finite domain as a quick lookup table for all slices of $\tau$.

Let us first introduce some operations on sets of partial functions used in $\mathbb{A}\langle X\rangle$ (only the informal intuition is given here; rigorous definitions can be found in [16]). Given sets of partial functions $\Theta, \Theta' \subseteq [X \rightarrow V]$, $\sqcup\Theta$ is the least informative partial function $\theta \in [X \rightarrow V]$ such that for any $\theta' \in \Theta$, $\theta' \sqsubseteq \theta$; $\max\Theta$ is the most informative $\theta \in \Theta$; $\Theta \sqcup \Theta' = \{\theta \sqcup \theta' \mid \theta \in \Theta, \theta' \in \Theta' \ s.t., \theta \sqcup \theta' \text{ exists}\}$; and $(\theta]_\Theta = \{\theta' \mid \theta' \in \Theta \text{ and } \theta' \sqsubseteq \theta\}$. Note that $\sqcup\Theta$ and $\max\Theta$ may not exist. Then Theorem 1 shows that, for any $\theta \in [X \rightarrow V]$, the trace slice $\tau{\restriction_\theta}$ is $\mathbb{T}(\max(\theta]_\Theta)$ after $\mathbb{A}\langle X\rangle$ processes $\tau$, where $\Theta$ is the domain of $\mathbb{T}$, calculated by $\mathbb{A}\langle X\rangle$ incrementally. Therefore, assuming that $\mathbb{A}\langle X\rangle$ is run on trace $\tau$, all one has to do in order to calculate a slice $\tau{\restriction_\theta}$ for a given $\theta \in [X \rightarrow V]$ is to calculate $\max(\theta]_\Theta$ followed by a lookup into $\mathbb{T}$. This way the trace $\tau$, which can be very long, is processed/traversed only once, as it is being generated, and appropriate data-structures are maintained by our algorithm that allow for retrieval of slices for any parameter instance $\theta$, without traversing $\tau$ again.

```
Algorithm 𝔸⟨X⟩
Input: parametric trace τ ∈ ℰ⟨X⟩*
Output: map 𝕋 ∈ [[X→V]→ℰ*] and
        set Θ ⊆ [X→V]
1  𝕋 ← ⊥; 𝕋(⊥) ← ϵ; Θ ← {⊥}
2  foreach e⟨θ⟩ in order in τ do
3  : foreach θ' ∈ {θ} ⊔ Θ do
4  : : 𝕋(θ') ← 𝕋(max (θ']_Θ) e
5  : endfor
6  : Θ ← {⊥, θ} ⊔ Θ
7  endfor
```

**Fig. 1.** Parametric slicing algorithm $\mathbb{A}\langle X\rangle$

Fig. 1 shows our trace slicing algorithm $\mathbb{A}\langle X\rangle$. In spite of $\mathbb{A}\langle X\rangle$'s small size, its proof of correctness is surprisingly intricate as shown in [16]. The algorithm $\mathbb{A}\langle X\rangle$ on input $\tau$, written more succinctly $\mathbb{A}\langle X\rangle(\tau)$, traverses $\tau$ from its first event to its last event and, for each encountered event $e\langle\theta\rangle$, updates both its data-structures, $\mathbb{T}$ and $\Theta$. After processing each event, the relationship between $\mathbb{T}$ and $\Theta$ is that the latter is the domain of the former. Line 1 initializes the data-structures: $\mathbb{T}$ is undefined everywhere (i.e., $\bot$) except for the undefined-everywhere function $\bot$, where $\mathbb{T}(\bot) = \epsilon$; as expected, $\Theta$ is then initialized to the set $\{\bot\}$. The code (lines 3 to 6) inside the outer loop (lines 2 to 7) can be triggered when a new event is received. When a new event is received, say $e\langle\theta\rangle$, $\mathbb{T}$ is updated as follows: for each $\theta' \in [X \rightarrow V]$ that can be obtained by combining $\theta$ with the compatible partial functions in the domain of the current $\mathbb{T}$, update $\mathbb{T}(\theta')$ by adding the non-parametric event $e$ to the end of the slice corresponding to the largest (i.e., most "knowledgeable") entry in the current table $\mathbb{T}$ that is less informative or as informative as $\theta'$; $\Theta$ is then extended in line 6.

**Example.** Consider the following sample parametric trace with events parametric in $\{a, b, c\}$: $\tau = e_1\langle a_1\rangle\ e_2\langle a_2\rangle\ e_3\langle b_1\rangle\ e_4\langle a_2 b_1\rangle\ e_5\langle a_1\rangle\ e_6\langle\rangle\ e_7\langle b_1\rangle$ . Table 1 shows how $\mathbb{A}\langle X\rangle$ works on $\tau$. An entry of the form "$\langle\theta\rangle : w$" in a table cell corresponding to a "current" parametric event $e\langle\theta\rangle$ means that $\mathbb{T}(\theta) = w$ after processing all the parametric events up to and including the current one; $\mathbb{T}$ is

**Table 1.** A run of the trace slicing algorithm $\mathbb{A}\langle X\rangle$

| $e_1\langle a_1\rangle$ | $e_2\langle a_2\rangle$ | $e_3\langle b_1\rangle$ | $e_4\langle a_2b_1\rangle$ | $e_5\langle a_1\rangle$ | $e_6\langle\rangle$ | $e_7\langle b_1\rangle$ |
|---|---|---|---|---|---|---|
| $\langle\rangle : \epsilon$ | $\langle\rangle : \epsilon$ | $\langle\rangle : \epsilon$ | $\langle\rangle : \epsilon$ | $\langle\rangle : \epsilon$ | $\langle\rangle : e_6$ | $\langle\rangle : e_6$ |
| $\langle a_1\rangle : e_1$ | $\langle a_1\rangle : e_1$ | $\langle a_1\rangle : e_1$ | $\langle a_1\rangle : e_1$ | $\langle a_1\rangle : e_1e_5$ | $\langle a_1\rangle : e_1e_5e_6$ | $\langle a_1\rangle : e_1e_5e_6$ |
| | $\langle a_2\rangle : e_2$ | $\langle a_2\rangle : e_2$ | $\langle a_2\rangle : e_2$ | $\langle a_2\rangle : e_2$ | $\langle a_2\rangle : e_2e_6$ | $\langle a_2\rangle : e_2e_6$ |
| | | $\langle b_1\rangle : e_3$ | $\langle b_1\rangle : e_3$ | $\langle b_1\rangle : e_3$ | $\langle b_1\rangle : e_3e_6$ | $\langle b_1\rangle : e_3e_6e_7$ |
| | | $\langle a_1b_1\rangle : e_1e_3$ | $\langle a_1b_1\rangle : e_1e_3$ | $\langle a_1b_1\rangle : e_1e_3e_5$ | $\langle a_1b_1\rangle : e_1e_3e_5e_6$ | $\langle a_1b_1\rangle : e_1e_3e_5e_6e_7$ |
| | | $\langle a_2b_1\rangle : e_2e_3$ | $\langle a_2b_1\rangle : e_2e_3e_4$ | $\langle a_2b_1\rangle : e_2e_3e_4$ | $\langle a_2b_1\rangle : e_2e_3e_4e_6$ | $\langle a_2b_1\rangle : e_2e_3e_4e_6e_7$ |

undefined on any other partial function. Obviously, the $\Theta$ corresponding to a cell is the union of all the $\theta$'s that appear in pairs "$\langle\theta\rangle : w$" in that cell. Trace slices for parameter instances, e.g., $\langle a_1b_1\rangle$ and $\langle a_2b_1\rangle$, which have *not* been seen in any observed event are also created. Note that, as each parametric event $e\langle\theta\rangle$ is processed, the non-parametric event $e$ is added at most once to each slice. □

$\mathbb{A}\langle X\rangle$ computes trace slices for all combinations of parameter instances observed in parametric trace events. Its complexity is therefore $O(n \times m)$ where $n$ is the length of the trace and $m$ is the number of all possible parameter combinations. However, $\mathbb{A}\langle X\rangle$ is not intended to be implemented directly; it is only used as a correctness backbone for other trace analysis algorithms, such as the monitoring algorithms discussed below. An alternative and apparently more efficient solution is to only record trace slices for parameter instances that actually appear in the trace (instead of for all combinations of them), and then construct the slice for a given parameter instance by combining such trace slices for compatible parameter instances. However, the complexity of constructing all possible trace slices at the end using such an algorithm is also $O(n \times m)$. In addition, $\mathbb{A}\langle X\rangle$ is more suitable for online monitoring: each event is sent to its slices (that are consumed by corresponding monitors) and never touched again.

$\mathbb{A}\langle X\rangle$ compactly and uniformly captures several special cases and subcases that are worth discussing. The discussion below can be formalized as an inductive (on the length of $\tau$) proof of correctness for $\mathbb{A}\langle X\rangle$, but we prefer to keep this discussion informal here and use it as a means to better explain the algorithm $\mathbb{A}\langle X\rangle$, providing the reader with additional intuition for its difficulty and compactness. A rigorous proof can be found in [16].

Let us first note that a partial function added to $\Theta$ will never be removed from $\Theta$; that's because $\Theta \subseteq \{\bot, \theta\} \sqcup \Theta$. The same holds true for the domain of $\mathbb{T}$, because line 4 can only add new elements to $\mathsf{Dom}(\mathbb{T})$; in fact, the domain of $\mathbb{T}$ is extended with precisely the set $\{\theta\} \sqcup \Theta$ after each event parametric in $\theta$ is processed by $\mathbb{A}\langle X\rangle$. Moreover, since $\mathsf{Dom}(\mathbb{T}) = \Theta = \Theta_\epsilon = \{\bot\}$ initially and since $\Theta \sqcup (\{\theta\} \sqcup \Theta) = \{\bot, \theta\} \sqcup \Theta$ while $\Theta_{\tau\, e\langle\theta\rangle} = \{\bot, \theta\} \sqcup \Theta_\tau$, where $\Theta_\tau$ is $\Theta$ after $\mathbb{A}\langle X\rangle$ processes $\tau$, we can inductively show that $\mathsf{Dom}(\mathbb{T}) = \Theta = \Theta_\tau$ each time after $\mathbb{A}\langle X\rangle$ is executed on a parametric trace $\tau$. Each $\theta'$ considered by the loop at lines 3-5 has the property that $\theta \sqsubseteq \theta'$, and at (precisely) one iteration of the loop $\theta'$ is $\theta$; indeed, $\theta \in \{\theta\} \sqcup \Theta$ because $\bot \in \Theta$. Essentially, the claimed Theorem 1 holds iff $\mathbb{T}(\theta') = \tau{\restriction_{\theta'}}$ after $\mathbb{T}(\theta')$ is updated in line 4. A tricky observation which is crucial for this is that the updates of $\mathbb{T}(\theta')$ do not interfere with each other for different $\theta' \in \{\theta\} \sqcup \Theta$; otherwise the non-parametric event $e$ may be added multiple times to some trace slices $\mathbb{T}(\theta')$.

Let us next informally argue, inductively, that it is indeed the case that $\mathbb{T}(\theta') = \tau\restriction_{\theta'}$ after $\mathbb{T}(\theta')$ is updated in line 4 (it vacuously holds on the empty trace). Since $\max(\theta']_\Theta \in \Theta$, the inductive hypothesis tells us that $\mathbb{T}(\max(\theta']_\Theta) = \tau\restriction_{\max(\theta']_\Theta}$; these are further equal to $\tau\restriction_{\theta'}$. Since $\theta \sqsubseteq \theta'$, the definition of trace slicing implies that $(\tau\, e\langle\theta\rangle)\restriction_{\theta'} = \tau\restriction_{\theta'}\, e$. Therefore, $\mathbb{T}(\theta')$ is indeed $(\tau\, e\langle\theta\rangle)\restriction_{\theta'}$ after line 4 of $\mathbb{A}\langle X\rangle$ is executed while processing the event $e\langle\theta\rangle$ that follows trace $\tau$. This concludes our informal proof sketch.

Let $\mathbb{A}\langle X\rangle(\tau).\mathbb{T}$ and $\mathbb{A}\langle X\rangle(\tau).\Theta$ be $\mathbb{T}$ and $\Theta$ of $\mathbb{A}\langle X\rangle$ after it processes $\tau$.

**Theorem 1.** *The following hold for any $\tau \in \mathcal{E}\langle X\rangle^*$:*

1. $\mathit{Dom}(\mathbb{A}\langle X\rangle(\tau).\mathbb{T}) = \mathbb{A}\langle X\rangle(\tau).\Theta = \Theta_\tau$;
2. $\mathbb{A}\langle X\rangle(\tau).\mathbb{T}(\theta) = \tau\restriction_\theta$ *for any* $\theta \in \mathbb{A}\langle X\rangle(\tau).\Theta$;
3. $\tau\restriction_\theta = \mathbb{A}\langle X\rangle(\tau).\mathbb{T}(\max(\theta]_{\mathbb{A}\langle X\rangle(\tau).\Theta})$ *for any* $\theta \in [X \rightharpoonup V]$.

## 3 Online Parametric Trace Monitoring

Here we first define monitors $M$ and parametric monitors $\Lambda X.M$. Like for parametric properties, which are just properties over parametric traces, we show that parametric monitors are also just monitors, but for parametric events and with instance-indexed states and output categories. We show that a parametric monitor $\Lambda X.M$ is a monitor for the parametric property $\Lambda X.P$, with $P$ the property monitored by $M$. Finally, we present an online monitoring algorithm based on algorithm $\mathbb{A}\langle X\rangle$ and then refine it to an efficient monitoring algorithm.

### 3.1 Monitors and Parametric Monitors

Non-parametric monitors are defined as a variant of Moore machines:

**Definition 7.** *A* ***monitor*** $M$ *is a tuple* $(S, \mathcal{E}, \mathcal{C}, \imath, \sigma : S \times \mathcal{E} \to S, \gamma : S \to \mathcal{C})$, *where $S$ is a set of states, $\mathcal{E}$ is a set of input events, $\mathcal{C}$ is a set of output categories, $\imath \in S$ is the initial state, $\sigma$ is the transition function, and $\gamma$ is the output function. The transition function is extended to $\sigma : S \times \mathcal{E}^* \to S$ the standard way.*

The notion of a monitor above is rather conceptual. Actual implementations of monitors need not generate all the state space apriori, but on a "by need" basis. Allowing monitors with infinitely many states is a necessity in our context. Even though only a finite number of states is reached during any given (finite) execution trace, there is, in general, no bound on how many. For example, monitors for context-free grammars like the ones in [15] have potentially unbounded stacks as part of their state. Also, as shown shortly, parametric monitors have domains of functions as state spaces, which are infinite as well.

**Definition 8.** $M = (S, \mathcal{E}, \mathcal{C}, \imath, \sigma, \gamma)$ *is a* ***monitor for property*** $P : \mathcal{E}^* \to \mathcal{C}$ *iff* $\gamma(\sigma(\imath, w)) = P(w)$ *for each* $w \in \mathcal{E}^*$. *Every monitor $M$ defines the property $\mathcal{P}_M : \mathcal{E}^* \to \mathcal{C}$ with $\mathcal{P}_M(w) = \gamma(\sigma(\imath, w))$; note that $M$ is a monitor for $\mathcal{P}_M$. Monitors $M$ and $M'$ are* ***equivalent****, written $M \equiv M'$ iff $\mathcal{P}_M = \mathcal{P}_{M'}$.*

**Proposition 1.** *Every property $P$ defines a monitor $\mathcal{M}_P$ with $\mathcal{M}_P$ a monitor for $P$. For any property $P$, $\mathcal{P}_{\mathcal{M}_P} = P$. For any monitor $M$, if $M = \mathcal{M}_P$ for some property $P$ then $\mathcal{M}_{\mathcal{P}_M} \equiv M$.*

We next define parametric monitors in the same style as the other parametric entities defined in this paper: starting with a base monitor and a set of parameters, the corresponding parametric monitor can be thought of as a set of base monitors running in parallel, one for each parameter instance.

**Definition 9.** *Given parameters $X$ with corresponding values $V$ and monitor $M = (S, \mathcal{E}, \mathcal{C}, \imath, \sigma : S \times \mathcal{E} \to S, \gamma : S \to \mathcal{C})$, the* ***parametric monitor*** *$\Lambda X.M$ is the monitor $([[X \rightarrow V] \to S], \mathcal{E}\langle X\rangle, [[X \rightarrow V] \to \mathcal{C}], \lambda\theta.\imath, \Lambda X.\sigma, \Lambda X.\gamma)$, with $\Lambda X.\sigma : [[X \rightarrow V] \to S] \times \mathcal{E}\langle X\rangle \to [[X \rightarrow V] \to S]$ and $\Lambda X.\gamma : [[X \rightarrow V] \to S] \to [[X \rightarrow V] \to \mathcal{C}]$ defined as*

$$(\Lambda X.\sigma)(\delta, e\langle\theta'\rangle)(\theta) = \begin{cases} \sigma(\delta(\theta), e) & \text{if } \theta' \sqsubseteq \theta \\ \delta(\theta) & \text{if } \theta' \not\sqsubseteq \theta \end{cases}$$
$$(\Lambda X.\gamma)(\delta)(\theta) = \gamma(\delta(\theta))$$

*for any $\delta \in [[X \rightarrow V] \to S]$ and any $\theta, \theta' \in [X \rightarrow V]$.*

Therefore, a state $\delta$ of parametric monitor $\Lambda X.M$ maintains a state $\delta(\theta)$ of $M$ for each parameter instance $\theta$, takes parametric events as input, and outputs categories indexed by parameter instances (one category of $M$ per instance).

**Proposition 2.** *If $M$ is a monitor for $P$, then $\Lambda X.M$ is a monitor for parametric property $\Lambda X.P$, or, $\mathcal{P}_{\Lambda X.M} = \Lambda X.\mathcal{P}_M$.*

### 3.2 Algorithm for Online Parametric Trace Monitoring

We next propose a monitoring algorithm for parametric properties. A first challenge here is how to represent the states of the parametric monitor. Inspired by algorithm $\mathbb{A}\langle X\rangle$, we encode functions $[[X \rightarrow V] \rightarrow S]$ as tables with entries indexed by parameter instances in $[X \rightarrow V]$ and with contents states in $S$. Such tables will have finite entries since each event binds only a finite number of parameters. Fig. 2 shows our monitoring algorithm for parametric properties. Given parametric property $\Lambda X.P$ and $M$ a monitor for $P$, $\mathbb{B}\langle X\rangle(M)$ yields a monitor that is equivalent to $\Lambda X.M$, that is, a monitor for $\Lambda X.P$. Section 4 shows one way to use this algorithm: a monitor $M$ is first synthesized from the base property $P$, then that monitor $M$ is used to synthesize the monitor $\mathbb{B}\langle X\rangle(M)$ for the parametric property $\Lambda X.P$.

$\mathbb{B}\langle X\rangle(M)$ follows very closely the algorithm for trace slicing in Fig. 1, the main difference being that trace slices are processed, as generated, by $M$: instead of calculating the trace slice of $\theta'$ by appending base event $e$ to the corresponding existing trace slice in line 4 of $\mathbb{A}\langle X\rangle$, we now calculate and store in table $\Delta$ the state of the "monitor instance" corresponding to $\theta'$ by sending $e$ to the corresponding existing monitor instance (line 4 in $\mathbb{B}\langle X\rangle(M)$); at the same time we also calculate the output corresponding to that monitor instance and store it in table $\Gamma$. In other words, we replace trace slices in $\mathbb{A}\langle X\rangle$ by local

monitors processing those slices. We also check whether $\Gamma(\theta')$ at line 5 violates or validates the property and, if so, a message including $\theta'$ is output. Given a monitor $M$, let $\mathcal{M}_{\mathbb{B}\langle X\rangle(M)}$ be the monitor defined by $\mathbb{B}\langle X\rangle(M)$. Theorem 2 then proves the correctness of $\mathbb{B}\langle X\rangle$.

**Theorem 2.** *$\mathcal{M}_{\mathbb{B}\langle X\rangle(M)} \equiv \Lambda X.M$ for any monitor $M$. If $M$ is a monitor for $P$, then $\mathcal{M}_{\mathbb{B}\langle X\rangle(M)}$ is a monitor for parametric property $\Lambda X.P$.*

```
Algorithm 𝔹⟨X⟩(M=(S,ℰ,𝒞,ι,σ,γ))
Input: parametric trace τ ∈ ℰ⟨X⟩*
Output:Γ : [[X→V]→𝒞] and Θ ⊆ [X→V]
1 Δ ← ⊥; Δ(⊥) ← ι; Θ ← {⊥}
2 foreach e⟨θ⟩ in order in τ do
3 : foreach θ' ∈ {θ} ⊔ Θ do
4 : : Δ(θ') ← σ(Δ(max(θ']_Θ), e)
5 : : Γ(θ') ← γ(Δ(θ'))
6 : endfor
7 : Θ ← {⊥, θ} ⊔ Θ
8 endfor
```

**Fig. 2.** Monitoring algorithm $\mathbb{B}\langle X\rangle$

### 3.3 Optimized Online Monitoring Algorithm

Algorithm $\mathbb{C}\langle X\rangle$ in Fig. 3 refines Algorithm $\mathbb{B}\langle X\rangle$ in Fig. 2 for efficient online monitoring. $\mathbb{C}\langle X\rangle$ essentially expands the body of the outer loop in $\mathbb{B}\langle X\rangle$ (lines 3 to 7 in Fig. 2). The direct use of $\mathbb{B}\langle X\rangle$ would yield prohibitive runtime overhead when monitoring large traces, because its inner loop requires search for all parameter instances in $\Theta$ that are compatible with $\theta$; this search can be very expensive. $\mathbb{C}\langle X\rangle$ introduces an auxiliary data structure and illustrates a mechanical way to accomplish the search, which also facilitates further optimizations.

```
Algorithm ℂ⟨X⟩(M = (S,ℰ,𝒞,ι,σ,γ))
Globals: mappings Δ : [[X→V]→S],
         Γ : [[X→V]→𝒞],
         𝒰 : [X→V] → 𝒫_f([X→V])
Initialization:
 𝒰(θ) ← ∅ for any θ ∈ [X→V],
 Δ(⊥) ← ι
function main(e⟨θ⟩)
 1 if Δ(θ)undefined then
 2 : foreach θ_max ⊏ θ (in reversed
   : topological order) do
 3 : : if Δ(θ_max) defined then
 4 : : : goto 7
 5 : : 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) undef. 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') ← σ(Δ(θ'), e)
18 : Γ(θ') ← σ(Δ(θ'))
19 endfor
function defineTo(θ, θ')
 1 Δ(θ) ← Δ(θ')
 2 foreach θ'' ⊏ θ do
 3 : 𝒰(θ'') ← 𝒰(θ'') ∪ {θ}
 4 endfor
```

**Fig. 3.** Monitoring algorithm $\mathbb{C}\langle X\rangle$

$\mathbb{C}\langle X\rangle$ uses three tables: $\Delta$, $\mathcal{U}$ and $\Gamma$. $\Delta$ and $\Gamma$ are the same as $\Delta$ and $\Gamma$ in $\mathbb{B}\langle X\rangle$, respectively. $\mathcal{U}$ is an auxiliary data structure used to optimize the search "for all $\theta' \in \{\theta\} \sqcup \Theta$" in $\mathbb{B}\langle X\rangle$ (line 3 in Fig. 2). It maps each parameter instance $\theta$ into the finite set of parameter instances encountered in $\Delta$ so far that are strictly more informative than $\theta$, i.e., $\mathcal{U}(\theta) = \{\theta' \mid \theta' \in \mathsf{Dom}(\Delta) \text{ and } \theta \sqsubset \theta'\}$. Another major difference between $\mathbb{B}\langle X\rangle$ and $\mathbb{C}\langle X\rangle$ is that $\mathbb{C}\langle X\rangle$ does *not* maintain $\Theta$ explicitly: $\Theta$ at the beginning/end of the body of the outer loop in $\mathbb{B}\langle X\rangle$ is $\mathsf{Dom}(\Delta)$ at the beginning/end of $\mathbb{C}\langle X\rangle$, respectively. However, $\Theta$ is fixed during the loop at lines 3 to 6 in $\mathbb{B}\langle X\rangle$ and updated atomically in line 7, while $\mathsf{Dom}(\Delta)$ can be changed at any time during the execution of $\mathbb{C}\langle X\rangle$.

$\mathbb{C}\langle X\rangle$ is composed of two functions, main and defineTo. The defineTo function takes two parameter instances, $\theta$ and $\theta'$, and adds a new entry corresponding to $\theta$ into $\Delta$ and $\mathcal{U}$. Specifically, it sets $\Delta(\theta)$ to $\Delta(\theta')$ and adds $\theta$ into the set $\mathcal{U}(\theta'')$ for each $\theta'' \sqsubset \theta$. The main function differentiates two cases when a new event $e\langle\theta\rangle$ is received and processed. The simpler case is that $\Delta$ is already defined on $\theta$, i.e., $\theta \in \Theta$ at the beginning of the iteration of the outer loop in $\mathbb{B}\langle X\rangle$. In this case, $\{\theta\} \sqcup \Theta = \{\theta' \mid \theta' \in \Theta \text{ and } \theta \sqsubseteq \theta'\} \subseteq \Theta$, so the lines 3 to 6 in $\mathbb{B}\langle X\rangle$ become precisely the lines 16 to 19 in $\mathbb{C}\langle X\rangle$. In the other case, when $\Delta$ is not already defined on $\theta$, main takes two steps to handle $e$. The first step searches for new parameter instances introduced by $\{\theta\} \sqcup \Theta$ and adds entries for them into $\Delta$ (lines 2 to 15). We first add an entry to $\Delta$ for $\theta$ at lines 2 to 7. Then we search for all parameter instances $\theta_{comp}$ that are compatible with $\theta$, making use of $\mathcal{U}$ (line 8 and 9); for each such $\theta_{comp}$, an appropriate entry is added to $\Delta$ for its lub with $\theta$, and $\mathcal{U}$ updated accordingly (lines 10 to 12). This way, $\Delta$ will be defined on all the new parameter instances introduced by $\{\theta\} \sqcup \Theta$ after the first step. In the second step, the related monitor states and outputs are updated in a similar way as in the first case (lines 16 to 19). It is interesting to note how $\mathbb{C}\langle X\rangle$ searches at lines 2 and 8 for $\max(\theta]_\Theta$ that $\mathbb{B}\langle X\rangle$ refers to at line 4 in Fig. 2: it enumerates all the $\theta_{max} \sqsubset \theta$ in reversed topological order (larger to smaller); we proved that $\max(\theta]_\Theta$ exists and this search will find it ([16]).

***Correctness of*** $\mathbb{C}\langle X\rangle$**.** We next argue informally the correctness of $\mathbb{C}\langle X\rangle$ (formal proofs can be found in [16]) by showing that it is equivalent to the body of the outer loop in $\mathbb{B}\langle X\rangle$. Suppose that parametric trace $\tau$ has already been processed by both $\mathbb{C}\langle X\rangle$ and $\mathbb{B}\langle X\rangle$, and a new event $e\langle\theta\rangle$ is to be processed next. First, note that $\mathbb{C}\langle X\rangle$ terminates: there is only a finite number of partial maps less informative than $\theta$, that is, only a finite number of iterations for the loops at lines 2 and 8 in main; since $\mathcal{U}$ is only updated at line 3 in defineTo, $\mathcal{U}(\theta)$ is finite for any $\theta \in [X \rightharpoonup V]$ and thus the loop at line 9 in main also terminates. Assuming that running the base monitor $M$ takes constant time, the worse case complexity of $\mathbb{C}\langle X\rangle(M)$ is $O(k \times l)$ to process $e\langle\theta\rangle$, where $k$ is $2^{|\mathsf{Dom}(\theta)|}$ and $l$ is the number of incompatible parameter instances in $\tau$. Parametric properties often have a fixed and small number of parameters, in which case $k$ is not significant. Depending on the trace, $l$ can grow arbitrarily large; in the worst case, each event may carry an instance incompatible with the previous ones.

Next result establishes the correctness of $\mathbb{C}\langle X\rangle$. Fix a monitor $M$. Let $\Delta^b_{\mathbb{C}}$ and $\Gamma^b_{\mathbb{C}}$ be the $\Delta_{\mathbb{C}}$ and $\Gamma_{\mathbb{C}}$ when $\mathsf{main}(e\langle\theta\rangle)$ in $\mathbb{C}\langle X\rangle(M)$ begins ("$b$"="begin"); let $\Delta^e_{\mathbb{C}}$ and $\Gamma^e_{\mathbb{C}}$ be the $\Delta_{\mathbb{C}}$ and $\Gamma_{\mathbb{C}}$ when $\mathsf{main}(e\langle\theta\rangle)$ ends ("$e$"="end"); let $\mathbb{B}\langle X\rangle(M)(\tau).\Delta$ and $\mathbb{B}\langle X\rangle(M)(\tau).\Gamma$ be the $\Delta$ and $\Gamma$ after $\mathbb{B}\langle X\rangle(M)$ processes trace $\tau$.

**Theorem 3.** *If* $\Delta^b_{\mathbb{C}} = \mathbb{B}\langle X\rangle(M)(\tau).\Delta$ *and* $\Gamma^b_{\mathbb{C}} = \mathbb{B}\langle X\rangle(M)(\tau).\Gamma$, *then* $\Delta^e_{\mathbb{C}} = \mathbb{B}\langle X\rangle(M)(\tau\, e\langle\theta\rangle).\Delta$ *and* $\Gamma^b_{\mathbb{C}} = \mathbb{B}\langle X\rangle(M)(\tau\, e\langle\theta\rangle).\Gamma$.

## 4 Implementation and Evaluation

We have implemented our online monitoring algorithm in a prototype, here called PMon (from "Parametric Monitoring"), and evaluated it on the DaCapo benchmark [6]. Some optimizations have also been implemented. Note that $\mathbb{C}\langle X\rangle$ iterates through all the possible parameter instances that are less informative than $\theta$ in three different loops: at lines 2 and 8 in main, and at line 2 in defineTo. Hence, it is important to reduce the number of such instances in each loop. A static analysis of the specification, discussed in [8], exhaustively explores all possible event combinations that can lead to violations of the property, and then the number of loop iterations is reduced by skipping parameter instances that cannot affect the result of monitoring. The static analysis is used at compile time to unroll the loops in $\mathbb{C}\langle X\rangle$ and reduce the size of $\Delta$ and $\mathcal{U}$.

Another optimization is based on the observation that the monitoring process needs to start only when certain events are received. Such events are called monitor creation events in [9]. The parameter instances carried by such creation events may also be used to reduce the number of parameter instances that need to be considered. An extreme, yet surprisingly common case is when creation events instantiate all the property parameters. In this case, the monitoring process does not need to search for compatible parameter instances even when an event with an incomplete parameter instance is observed. The current JavaMOP [9] supports only traces whose monitoring starts with a fully instantiated creation event; this was perceived as an inherent limitation of JavaMOP, a consequence of its generality [3]. Interestingly, it now becomes just a common-case optimization of our novel, general and unrestricted technique presented here.

***Experiments and Evaluation.*** The following properties from [8] were checked in our experiments. The latter two cannot be handled by JavaMOP.

- LeakingSync. Only access a synchronized collection using its synchronized wrapper. One violation pattern monitored: $\Lambda c.\ \mathsf{sync}(c)\ \mathsf{asyncAccess}(c)$.
- FailSafeEnum. Do not update a vector while enumerating over it. The following violation pattern monitored: $\Lambda v, e.\ \mathsf{createEnum}(v, e)\ \mathsf{modify}(v)\ \mathsf{access}(e)$.
- ASyncIterCol. Only iterate a synchronized collection $c$ when holding a lock on $c$. Two violation patterns monitored: $\Lambda c, i.\ \mathsf{sync}(c)\ \mathsf{asyncCreateIter}(c, i)$ and $\Lambda c, i.\ \mathsf{sync}(c)\ \mathsf{syncCreateIter}(c, i)\ \mathsf{asyncAccess}(i)$.
- ASyncIterMap. Only iterate a synchronized map $m$ when holding a lock on $m$. Two violation patterns monitored: $\Lambda m, s, i.\ \mathsf{sync}(m)\ \mathsf{getSet}(m, s)\mathsf{asyncCreateIter}(s, i)$, $\Lambda m, s, i.\ \mathsf{sync}(m)\ \mathsf{getSet}(m, s)\ \mathsf{syncCreateIter}(s, i)\ \mathsf{asyncAccess}(i)$.

**Table 2.** Average percent runtime overhead for PMon, manually coded monitors(Man), Tracematches(TM) and JavaMOP(MOP), with convergence within 3%. *: Cannot be handled by JavaMOP

| | LeakingSync | | | | FailSafeEnum | | | | ASyncIterCol* | | | ASyncIterMap* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | PMon | Man | TM | MOP | PMon | Man | TM | MOP | PMon | Man | TM | PMon | Man | TM |
| antlr | 2 | 1 | 5 | 2 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 2 | 1 | 2 |
| bloat | 140 | 145 | 785 | 141 | 0 | 2 | 0 | 2 | 721 | 150 | 1459 | 660 | 164 | 2300 |
| chart | 25 | 21 | 70 | 24 | 1 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 |
| eclipse | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 1 | 2 | 0 | 1 | 0 | 0 |
| fop | 53 | 47 | 146 | 50 | 1 | 0 | 0 | 0 | 2 | 1 | 1 | 2 | 1 | 0 |
| hsqldb | 2 | 5 | 24 | 4 | 0 | 0 | 25 | 0 | 1 | 0 | 25 | 1 | 0 | 0 |
| jython | 62 | 52 | 55 | 59 | 0 | 0 | 8 | 0 | 0 | 0 | 9 | 0 | 0 | 9 |
| luindex | 8 | 7 | 20 | 7 | 7 | 4 | 16 | 3 | 3 | 0 | 2 | 1 | 0 | 4 |
| lusearch | 12 | 10 | 52 | 12 | 5 | 2 | 28 | 7 | 4 | 1 | 9 | 0 | 0 | 8 |
| pmd | 55 | 47 | 53 | 52 | 0 | 0 | 0 | 0 | 37 | 30 | 36 | 50 | 49 | 53 |
| xalan | 39 | 29 | 117 | 40 | 5 | 6 | 33 | 4 | 1 | 1 | 6 | 1 | 1 | 7 |

These properties were chosen since they involve some of the most used data structures in Java and generate intensive monitoring overhead; also, their overhead is a consequence of the huge number of parameter instances to handle and *not* because of the complexity of the base, non-parametric properties.

Using the above properties, we compared our implementation with three other monitoring approaches, namely: optimal manually implemented monitors[1], Tracematches and JavaMOP. The latter two are chosen for comparison because they are very efficient monitoring systems [3,9]. The evaluation carried out on a 1.5GB, Pentium 4 2.66GHz machine running Ubuntu 7.10. We used the DaCapo benchmark version 2006-10; it contains eleven open source programs, as shown in Table 2. The provided default input was used with the -converge option to execute the benchmark multiple times until the execution time falls within 3% variation. The average execution time of six iterations after convergence is used.

The results are shown in Table 2. Among all 44 experiments, PMon generates 14% runtime overhead on average with more than 15% in only 10 experiments, showing that our algorithm is efficient[2]. Comparing with other approaches, we have the following observations: 1) PMon performed as well as or better than Tracematches in all cases, although the latter has domain specific optimizations for its hard-wired parametric regular patterns; 2) PMon generates similar runtime overhead as JavaMOP in the cases that can be handled by JavaMOP, showing that PMon conservatively extends the limited algorithm implemented in JavaMOP; 3) the monitoring code generated by PMon performs as well as the manually implemented monitors in most cases in the evaluated properties.

## 5 Related Work

Several approaches have been proposed to specify and monitor parametric properties. Tracematches [1,3] is an extension of AspectJ [2] supporting specifications

[1] Borrowed from [8], supposedly the best monitoring code for the given properties.

[2] These properties generated a tremendous number of events and parameter instances, e.g., millions of events and instances seen for LeakingSync and FailSafeEnum [9].

of parametric regular patterns; when patterns are matched during the execution, user-defined advice can be triggered. J-LO [7] is a variation of Tracematches that supports linear temporal logic properties. Also based on AspectJ, [13] proposes Live Sequence Charts (LSC) [11] as an inter-object scenario-based specification formalism; LSC is implicitly parametric, requiring dynamic parameter binding at runtime. Tracematches, J-LO and LSC [13] support a limited number of parameters, and each handles parameterization in a way that is specific to its particular specification formalism. Our proposed technique is generic in the specification formalism, and admits a potentially unlimited number of parameters.

Program Query Language (PQL) [14] allows the specification and monitoring of parametric context-free grammar (CFG) patterns. Unlike previous approaches, PQL can associate parameters with sub-patterns that can be recursively matched at runtime, yielding a potentially unbounded number of parameters. PQL's approach to parametric monitoring is specific to its particular CFG-based specification formalism. Also, PQL's design does not support arbitrary execution traces. For example, field updates and method begins are not observable. Like PQL, our technique also allows an unlimited number of parameters. Unlike PQL, our technique is not limited to particular events and is generic in the property specification formalism; CFGs are just one such possible formalism.

Eagle [4], RuleR [5], and Program Trace Query Language (PTQL) [12] are very general trace specification and monitoring systems, whose specification formalisms allow complex properties with parameter bindings anywhere in the specification. Eagle and RuleR are based on fixed-point logics and rewrite rules, while PTQL is based on SQL relational queries. These systems attempt to define general specification formalisms supporting data binding among many other features, while we attempt to define a general parameterization approach that is logic-independent. The very general specification formalisms tend to be slower [3,15,9]. We believe that our techniques can be used as an optimization for certain common types of properties expressible in these systems: use any of these to specify the base property $P$, then use our generic techniques to analyze $\Lambda X.P$.

## 6 Concluding Remarks and Future Work

A semantic foundation for parametric traces, properties and monitoring was proposed. A parametric trace slicing technique, which was discussed and proved correct, allows the extraction of all the non-parametric trace slices from a parametric slice by traversing the original trace only once and dispatching each parametric event to its corresponding slices. It thus enables the leveraging of any non-parametric, i.e., conventional, trace analysis techniques to the parametric case. A parametric monitoring technique then makes use of it to monitor arbitrary parametric properties against parametric execution traces using and indexing ordinary monitors for the base, non-parametric property. An implementation of the discussed techniques reveals that their generality, compared to the existing similar, but limited, techniques, does not come at a performance expense.

## References


1. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhotak, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: OOPSLA 2005. ACM, New York (2005)
2. AspectJ, `http://eclipse.org/aspectj/`
3. Avgustinov, P., Tibble, J., de Moor, O.: Making trace monitoring feasible. In: OOPSLA 2007. ACM, New York (2007)
4. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 44–57. Springer, Heidelberg (2004)
5. Barringer, H., Rydeheard, D.E., Havelund, K.: Rule systems for run-time monitoring: From Eagle to RuleR. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 111–125. Springer, Heidelberg (2007)
6. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., Van Drunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA 2006. ACM Press, New York (2006)
7. Bodden, E.: J-lo, a tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University (2005)
8. Bodden, E., Chen, F., Roşu, G.: Dependent advice: A general approach to optimizing history-based aspects. In: AOSD 2009. ACM, New York (2009)
9. Chen, F., Roşu, G.: MOP: An Efficient and Generic Runtime Verification Framework. In: OOPSLA 2007. ACM, New York (2007)
10. Chen, F., Roşu, G.: Mining Parametric State-Based Specifications from Executions. Technical Report UIUCDCS-R-2008-3000, Dept. of Computer Science at UIUC (2008)
11. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. Formal Methods in System Design 19(1), 45–80 (2001)
12. Goldsmith, S., O'Callahan, R., Aiken, A.: Relational queries over program traces. In: OOPSLA 2005. ACM Press, New York (2005)
13. Maoz, S., Harel, D.: From multi-modal scenarios to code: compiling lscs into aspectj. In: FSE 2006, pp. 219–230. ACM, New York (2006)
14. Martin, M., Livshits, V.B., Lam, M.S.: Finding application errors and security flaws using PQL: a program query language. In: OOPSLA 2005. ACM, New York (2005)
15. Meredith, P., Jin, D., Chen, F., Roşu, G.: Efficient monitoring of parametric context-free patterns. In: ASE 2008. IEEE/ACM (2008)
16. Roşu, G., Chen, F.: Parametric Trace Slicing and Monitoring. Technical Report UIUCDCS-R-2008-2977, Dept. of Computer Science at UIUC (2008)