University of Illinois at Urbana-Champaign
Department of Computer Science UIUCDCS-R-2008-29

Parametric Trace Slicing and Monitoring

Grigore Roşu and Feng Chen

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

Abstract

Trace analysis plays a fundamental role in many program analysis approaches, such as runtime verification, testing, monitoring, and specification mining. Recent research efforts bring empirical evidence that execution traces are frequently comprised of many meaningful trace slices merged together, each slice corresponding to instances of relevant parameters. For example, a Java program creating iterators i_1 and i_2 over vector v_1 may yield a trace "$\mathsf{create}\langle v_1\, i_1\rangle$ $\mathsf{next}\langle i_1\rangle$ $\mathsf{create}\langle v_1\, i_2\rangle$ $\mathsf{update}\langle v_1\rangle$ $\mathsf{next}\langle i_1\rangle$" parametric in vector v and iterator i, whose slices corresponding to parameter instances "$v, i \mapsto v_1, i_1$" and "$v, i \mapsto v_1, i_2$" are "$\mathsf{create}\langle v_1\, i_1\rangle$ $\mathsf{next}\langle i_1\rangle$ $\mathsf{update}\langle v_1\rangle$ $\mathsf{next}\langle i_1\rangle$" and, respectively, "$\mathsf{create}\langle v_1\, i_2\rangle$ $\mathsf{update}\langle v_1\rangle$ ". Several current trace analysis techniques and systems allow the specification of parametric properties, and the analysis of execution traces with respect to each instance of the parameters. However, the current solutions have limitations: some in the specification formalism, others in the type of trace they support; moreover, they share common notions, intuitions, even techniques and algorithms, suggesting that a fundamental study and 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 property or trace 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 technique, by applying the later on each trace slice. As an instance, a parametric property monitoring technique is then presented, which processes each trace slice online. Thanks to the generality of parametric trace slicing, the parametric property monitoring technique reduces to encapsulating and indexing unrestricted and well-understood non-parametric property monitors (e.g., finite or push-down automata).

The presented parametric trace slicing and monitoring techniques have been implemented and extensively evaluated. Measurements of runtime overhead confirm that the generality of the discussed techniques does not come at a performance expense when compared with existing parametric trace monitoring systems.

1. Introduction and Motivation

Parametric traces 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. Besides its name, each event of a parametric trace carries a partial instance of property parameters of interest; for example, if one is interested in analyzing vectors and iterators in Java, then execution traces of interest may contain events "$\mathsf{create}\langle v\, i\rangle$" (iterator i is created for vector v), "$\mathsf{update}\langle v\rangle$" ($v$ is modified), and "$\mathsf{next}\langle i\rangle$" ($i$ is accessed using its next element method), instantiated for particular vector and iterator instances. Most properties of parametric traces are also parametric, i.e., refer to each particular parameter instance; for our example, a property may be "vectors are not allowed to change while accessed through iterators", which is parametric in a vector 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 our example, the parametric property expressed as a regular expression (here matches mean violations) can be[1] "$\Lambda v, i.\ \mathsf{create}\langle v\, i\rangle\ \mathsf{next}\langle i\rangle^*\ \mathsf{update}\langle v\rangle^+\ \mathsf{next}\langle i\rangle$".

This paper addresses the problem of parametric trace analysis from a foundational perspective: Given a parametric trace τ and a parametric property $\Lambda X.P$, what does it mean for τ to be a good or a bad trace for $\Lambda X.P$? How can we show it? How can we leverage, to the parametric case, our knowledge and techniques to analyze conventional, non-parametric traces against conventional, non-parametric properties? In this paper we first formulate and then rigorously answer and empirically validate our answer to these questions, in the context of runtime verification. In doing so, a technique for trace slicing is also presented and shown correct, which we regard as *the* central result in parametric trace analysis.

Our concrete contributions are explained after the related work.

Related work. Several approaches have been proposed to specify and monitor parametric properties. Tracematches [1, 3] is an extension of AspectJ [2] allowing specification 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; the user provided actions are executed when the temporal properties are violated. Also based on AspectJ, [12] proposes Live Sequence Charts (LSC) [10] as an inter-object scenario-based specification formalism; LSC is implicitly parametric, requiring dynamic parameter binding at runtime. Tracematches, J-LO and LSC [12] support a limited number of parameters, and each has its own approach to handle parameterization specific to its particular specification formalism. On the contrary, our proposed technique is generic in the specification formalism, and admits a potentially unlimited number of parameters.

JavaMOP [14, 9] is a parametric specification and monitoring system for Java that is generic in the specification formalism for base properties, each formalism being included as a logic plugin. Monitoring code is generated from parametric specifications and woven within the original Java program, also using AspectJ, but using a different approach that allows it to encapsulate monitors for non-parametric properties as blackboxes. Unfortunately, JavaMOP's genericity comes at a price: it can only monitor execution traces in which the first event in each slice instantiates all the property parameters. This limitation prevents JavaMOP from monitoring some basic parametric properties, including ones discussed in this paper. Our novel approach to parametric trace slicing and monitoring proposed in this paper does not have JavaMOP's limitation.

[1] From here on we omit writing the event parameters in parametric properties, because they are redundant. E.g., $\Lambda v, i.$ $\mathsf{create}\ \mathsf{next}^*\ \mathsf{update}^+\ \mathsf{next}$.

Program Query Language (PQL) [13] allows the specification and monitoring of parametric context-free grammar (CFG) patterns. Unlike previous approaches that only allow a fixed (finite) number of property parameters, 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 now support arbitrary execution traces. For example, field updates and method begins are not observable; to circumvent the latter, PQL allows for observing traces local to method calls. 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) [11] are very general trace specification and monitoring systems, whose specification formalisms allow complex properties with parameter bindings anywhere in the specification (not only at the beginning, like we do). Eagle and RuleR are based on fixed-point logics and rewrite rules, while PTQL is based on SQL relational queries. These systems tackle a different aspect of generality than we do: they 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. As discussed in [3, 14, 9] (Eagle and PQL cases), the very general specification formalisms tend to be slower; this is not surprising, as the more general the formalism the less the potential for optimizations. 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 $\Lambda X.P$.

Since our proposed techniques yield performance comparable to hand-optimized monitors (see Section 9), at the same time being more general and overcoming the limitations of the current techniques supporting parameter binding only at the top of their properties, we believe that parametric properties as defined in this paper are perhaps a "sweet spot" of runtime verification, in the sense that they are expressive enough, yet quite efficiently monitorable.

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, Theorem 1, proves our parametric trace slicing technique correct, positively answering the following question: given a parametric execution trace τ, can one effectively find the slices $\tau{\restriction}_\theta$ corresponding to each parameter instance θ without having to traverse the trace for each θ? Our second result, Theorem 2, proves our parametric monitoring technique correct, which positively answers the following question: is it possible to monitor arbitrary parametric properties $\Lambda X.P$ against parametric execution traces τ, provided that the root non-parametric property P is monitorable using conventional monitors? Finally, our implementation answers positively the following question: can we implement a general purpose parametric property monitoring tool comparable in performance with existing parametric property monitoring tools, which currently work with restricted types of properties or traces?

Paper structure. Section 2 formalizes parametric events, traces and properties, and defines trace slicing. Section 3 discusses examples of parametric properties. Section 4 introduces mathematical nations. Section 5 establishes a tight connection between the parameter instances in a trace and the parameter instance used for slicing. Sections 6 to 8 discuss our main techniques for parametric trace slicing and monitoring, and prove them correct. Section 9 discusses our implementation of these techniques and its evaluation.

2. Parametric Traces and Properties

Here we introduce the notions of event, trace and property, first non-parametric and then parametric. Traces are sequences of events. Parametric events can carry data-values, as instances of parameters. Parametric traces are traces over parametric events. Properties are trace classifiers, that is, mappings partitioning the space of traces into categories (violating traces, validating traces, don't know traces etc.). Parametric properties are parametric trace classifiers and provide, for each parameter instance, the category to which the trace slice corresponding to that parameter instance belongs. Trace slicing is defined as a reduct operation that forgets all the events that are unrelated to the given parameter instance.

2.1 The Non-Parametric Case

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 $e \in w$.*

Our parametric trace slicing and monitoring techniques in Sections 6 and 8 can be easily adapted to also work with infinite traces. Since infinite versus finite traces is not an important aspect of the work reported here, we keep the presentation free of unnecessary technical complications and consider only finite traces.

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 $\mathcal{E} = \{\mathsf{acquire}, \mathsf{release}, \mathsf{begin}, \mathsf{end}\}$ and execution traces corresponding to this resource are sequences of the form "begin acquire acquire release end begin end", "begin acquire acquire", "begin acquire release acquire end", etc. For now there are no "good" or "bad" execution traces. □

There is a plethora of formalisms to specify trace requirements. Many of these result in specifying at least two types of traces: those *validating* the specification (i.e, correct traces), and those *violating* the specification (i.e., incorrect traces).

Example. *(part 2)* Consider a regular expression specification, $(\mathsf{begin}(\epsilon \mid (\mathsf{acquire}(\mathsf{acquire} \mid \mathsf{release})^*\mathsf{release}))\mathsf{end})^*$, 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. Assume that the resource can be acquired and released multiple times, with the effect of acquiring and respectively releasing it precisely once; regular expressions cannot specify matched acquire/release events, we are going to do so using context-free patterns in the next section. The validating traces for this property are those satisfying 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". □

In general, traces are not enforced to correspond to terminated programs (this is particularly useful in monitoring); if one wants to enforce traces to correspond to terminated programs, then one can have the system generate a special "end-of-trace" event and have the property specification require that event at the end of each trace.

Therefore, a trace property may partition the space of traces into more than two categories. For some specification formalisms, for example ones based on fuzzy logics or multiple truth values, the set of traces may be split into more than three categories, even into a continuous space of categories.

DEFINITION 2. *An $\mathcal{E}$-**property** P, or simply a (base or non-parametric) **property**, is a function $P : \mathcal{E}^* \rightarrow \mathcal{C}$ partitioning the set of traces into categories $\mathcal{C}$. It is common, though 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.*

We believe that the definition of non-parametric trace property above is general enough that it can easily accommodate any particular specification formalism, such as ones based on linear temporal logics, regular expressions, context-free grammars, etc. All one needs to do in order to instantiate the general results in this paper for a particular specification formalism is to decide upon the desired categories in which traces are intended to be classified, and then define the property associated to a specification accordingly.

For example, if the specification formalism of choice is that of regular expressions over $\mathcal{E}$ and one is interested in classifying traces in three categories as in our example above, then one can pick $\mathcal{C}$ to be the set $\{\mathsf{validating}, \mathsf{violating}, \mathsf{don't\ know}\}$ and, for a given regular expression E, define its associated property $P_E : \mathcal{E}^* \rightarrow \mathcal{C}$ as follows: $P_E(w) = \mathsf{validating}$ iff w is in the language of E, $P_E(w) = \mathsf{violating}$ iff there is no $w' \in \mathcal{E}^*$ such that $w\,w'$ is in the language of E, and $P_E(w) = \mathsf{don't\ know}$ otherwise; this is the monitoring semantics of regular expressions in JavaMOP [9].

Other semantic choices are possible even for the simple case of regular expressions; for example, one may choose $\mathcal{C}$ to be the set $\{\mathsf{matching}, \mathsf{don't\ care}\}$ and define $P_E(w) = \mathsf{matching}$ iff w is in the language of E, and $P_E(w) = \mathsf{don't\ care}$ otherwise; this is the semantics of regular expressions in Tracematches [1], where, depending upon how one writes the regular expression, matching can mean either a violation or a validation of the desired property.

In some applications, one may not be interested in certain categories of traces, such as in those classified as don't know or don't care; if that is the case, then those applications can simply ignore these, like Tracematches and JavaMOP do. It may be worth making it explicit that in this paper we do not attempt to propose or promote any particular formalism for specifying properties about execution traces. Instead, our approach is to define properties as generally as possible to capture the various specification formalisms that we are aware of as special cases, and then to develop our subsequent techniques to work with such general properties.

An additional benefit of defining properties so generally, as mappings from traces to categories, is that parametric properties, in spite of their much more general flavor, are also properties (but, obviously, over different traces and over different categories).

2.2 The Parametric Case

Events often carry concrete data instantiating abstract parameters.

Example. *(part 3)* In our running example, events acquire and release are parametric in the resource being acquired or released; 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 $\mathsf{acquire}\langle r \mapsto r_1\rangle$, $\mathsf{release}\langle r \mapsto r_2\rangle$, 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. □

We let $[A \rightarrow B]$ and $[A \overset{\circ}{\rightarrow} B]$ denote the sets of total and respectively partial functions from A to B.

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

Therefore, a parametric event is an event carrying values for zero, one, several or even all the parameters, and a parametric trace is a finite sequence of parametric events. In practice, the number of values carried by an event is finite; however, we do not need to enforce this restriction in our theoretical developments. Also, in practice the parameters may be typed, in which case the set of their corresponding values is given by their type. To simplify writing, we occasionally assume the set of parameter values V_X implicit.

Example. *(part 4)* A parametric trace for our running example can be the following: $\mathsf{begin}\langle\bot\rangle$ $\mathsf{acquire}\langle\theta_1\rangle$ $\mathsf{acquire}\langle\theta_2\rangle$ $\mathsf{acquire}\langle\theta_1\rangle$ $\mathsf{release}\langle\theta_1\rangle$ $\mathsf{end}\langle\bot\rangle$ $\mathsf{begin}\langle\bot\rangle$ $\mathsf{acquire}\langle\theta_2\rangle$ $\mathsf{release}\langle\theta_2\rangle$ $\mathsf{end}\langle\bot\rangle$, where θ_1 maps r to r_1 and θ_2 maps r to r_2. To simplify writing, we take the freedom to only list the parameter instance values when writing parameter instances, that is, $\langle r_1\rangle$ instead of $\langle r \mapsto r_1\rangle$, or $\tau\upharpoonright_{r_2}$ instead of $\tau\upharpoonright_{r \mapsto r_2}$, etc. With this notation, the above trace becomes: $\mathsf{begin}\langle\rangle$ $\mathsf{acquire}\langle r_1\rangle$ $\mathsf{acquire}\langle r_2\rangle$ $\mathsf{acquire}\langle r_1\rangle$ $\mathsf{release}\langle r_1\rangle$ $\mathsf{end}\langle\rangle$ $\mathsf{begin}\langle\rangle$ $\mathsf{acquire}\langle r_2\rangle$ $\mathsf{release}\langle r_2\rangle$ $\mathsf{end}\langle\rangle$. This trace involves two resources, r_1 and r_2, and it really consists of *two trace slices*, one for each resource, merged together. The begin and end events belong to both trace slices. The slice corresponding to θ_1 is "begin acquire acquire release end begin end", while the one for θ_2 is "begin acquire end begin acquire release end". □

DEFINITION 4. ***(Trace slicing)** Given parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$ and partial function θ in $[X \overset{\circ}{\rightarrow} V_X]$, we let the θ-**trace slice** $\tau\upharpoonright_\theta \in \mathcal{E}^*$ be the non-parametric trace in $\mathcal{E}^*$ defined as follows:*

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

where $\theta' \sqsubseteq \theta$ iff for any $x \in X$, if $\theta'(x)$ is defined then $\theta(x)$ is also defined and $\theta'(x) = \theta(x)$.

Therefore, the trace slice $\tau\upharpoonright_\theta$ first filters out all the parametric events that are not relevant for the instance θ, i.e., which contain instances of parameters that θ does not care about, and then, for the remaining events relevant to θ, it forgets the parameters so that the trace can be checked against base, non-parametric properties.

Specifying properties over parametric traces is rather challenging, because one may want to specify a property for one generic parameter instance and then say "and so on for all the other instances". In other words, one may want to specify a universally quantified property over base events, but, unfortunately, the underlying specification formalism may not allow universal quantification over data; for example, none of the conventional formalisms to specify properties on linear traces listed above (i.e, linear temporal logics, regular expressions, context-free grammars) or mentioned in the rest of the paper has universal data quantification.

DEFINITION 5. *Let X be a set of parameters together with their corresponding parameter values V_X, like in Definition 3, and let $P : \mathcal{E}^* \rightarrow \mathcal{C}$ 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 $[[X \overset{\circ}{\rightarrow} V_X] \rightarrow \mathcal{C}]$)*

$$\Lambda X.P : \mathcal{E}\langle X\rangle^* \rightarrow [[X \overset{\circ}{\rightarrow} V_X] \rightarrow \mathcal{C}]$$

defined as $(\Lambda X.P)(\tau)(\theta) = P(\tau\upharpoonright_\theta)$ for any $\tau \in \mathcal{E}\langle X\rangle^*$ and any $\theta \in [X \overset{\circ}{\rightarrow} V_X]$. 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_φ is defined using a pattern or formula φ 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 $[[X \overset{\circ}{\to} V_X] \to \mathcal{C}]$, i.e., 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.

Example. *(part 5)* Let P be the non-parametric property specified by the regular expression in the second part of our running example above (using the mapping of regular expressions to properties discussed in the second part of our running example and after Definition 2 – i.e., the JavaMOP semantic approach to parametric monitoring [9]). Since we want P to hold for any resource instance, we then define the following parametric property (i.e., $\Lambda r.P$):

$$\Lambda r.(\mathsf{begin}\ (\epsilon \mid (\mathsf{acquire}\ (\mathsf{acquire} \mid \mathsf{release})^*\ \mathsf{release}))\ \mathsf{end})^*.$$

If τ is the parametric trace and θ_1 and θ_2 are the parameter instances in the fourth part of our running example, then the semantics of the parametric property above on trace τ is $\mathsf{validating}$ for parameter instance θ_1 and $\mathsf{violating}$ for parameter instance θ_2. □

3. Examples of Parametric Properties

The simple example in Section 2.2 introduced parametric traces and properties, showing that events can instantiate one or no parameter. In this section we discuss other examples of parametric properties, defined using trace specification formalisms such as linear temporal logics and context free grammars or variants of these, some with more than one parameter and some with more than validating and violating trace categories, even with an infinite number of categories. For each of the examples, we give hints on how our subsequent techniques in Sections 6 and 8 work.

3.1 Authenticate before use

Consider a server authenticating and using keys, say k_1, k_2, k_3, etc., whose execution traces contain events of the form $\mathsf{authenticate}\langle k \mapsto k_1\rangle$, $\mathsf{use}\langle k \mapsto k_2\rangle$, etc., written more compactly $\mathsf{authenticate}\langle k_1\rangle$, $\mathsf{use}\langle k_2\rangle$, etc. A possible trace of such a system can be $\tau = \mathsf{authenticate}\langle k_1\rangle\ \mathsf{authenticate}\langle k_3\rangle\ \mathsf{use}\langle k_3\rangle\ \mathsf{use}\langle k_2\rangle\ \mathsf{authenticate}\langle k_2\rangle\ \mathsf{use}\langle k_1\rangle\ \mathsf{use}\langle k_2\rangle\ \mathsf{use}\langle k_3\rangle$. A parametric property for such a system can be "each key must be authenticated before use", which, using linear temporal logic as a specification formalism for the corresponding base property, can be expressed as

$$\Lambda k.\Box(\mathsf{use} \to \diamond\mathsf{authenticate}).$$

Such parametric LTL properties can be expressed in both J-LO [7] and JavaMOP [9] (the later using its LTL logic plugin). For the trace above and θ_3 the instance k_3 of k, the sliced trace $\tau{\restriction}_{\theta_3}$ is the trace "$\mathsf{authenticate\ use\ use}$" corresponding to the parametric subtrace "$\mathsf{authenticate}\langle k_3\rangle\ \mathsf{use}\langle k_3\rangle\ \mathsf{use}\langle k_3\rangle$" of events relevant to θ_3 in τ, but keeping only the base events; also, if θ_2 is the k_2 instance of k then $\tau{\restriction}_{\theta_2}$ is the trace "$\mathsf{use\ authenticate\ use}$". Our trace slicing algorithm discussed in Section 6 processes the parametric trace only once, traversing it from the first parametric event to the last, incrementally calculating several meaningful trace slices so that it can instantaneously report the slice for any parameter instance when asked. Using, e.g., the finite trace semantics for LTL in [17], $\Lambda k.\Box(\mathsf{use} \to \diamond\mathsf{authenticate})(\tau)(k \mapsto k_3) = \mathsf{true}$ and $\Lambda k.\Box(\mathsf{use} \to \diamond\mathsf{authenticate})(\tau)(k \mapsto k_2) = \mathsf{false}$. Our parametric monitoring algorithm in Section 8 reports a violation for instance $k \mapsto k_2$ precisely when the first $\mathsf{use}\langle k_2\rangle$ is encountered.

3.2 Safe iterators

Consider the following property for iterators created over vectors: when an iterator is created for a vector, one is not allowed to modify the vector while its elements are traversed using the iterator (this property is actually checked by Java 5 at runtime and an exception thrown if violated). Supposing that parametric event $\mathsf{create}\langle v\, i\rangle$ is generated when iterator i is created for vector v, $\mathsf{update}\langle v\rangle$ is generated whenever v is modified, and $\mathsf{next}\langle i\rangle$ is generated when i is accessed using its "next element" interface, then one can write it as the parametric regular expression property

$$\Lambda v, i.\mathsf{create}\ \mathsf{next}^*\ \mathsf{update}^+\ \mathsf{next}.$$

Such parametric regular expression properties can be expressed in both Tracematches [1] and JavaMOP [9] (the latter using its ERE plugin). We here assumed that the matching of the regular expression corresponds to violation of the base property (recall that one can give different property semantics to regular expressions). Thus, the parametric property is violated by a given trace and a given parameter instance whenever the regular pattern above is matched by the corresponding trace slice. For example, if $\tau = \mathsf{create}\langle v_1\, i_1\rangle\ \mathsf{next}\langle i_1\rangle\ \mathsf{create}\langle v_1\, i_2\rangle\ \mathsf{update}\langle v_1\rangle\ \mathsf{next}\langle i_1\rangle$ is a parametric trace where two iterators are created for a vector, then $\tau{\restriction}_{v_1\, i_1} = \mathsf{create\ next\ update\ next}$ and $\tau{\restriction}_{v_1\, i_2} = \mathsf{create\ update}$, so τ violates the parametric property (i.e., matches the regular pattern above) on instance "$v_1\, i_1$", but not on instance "$v_1\, i_2$". Note that in this example there are more than two parameters in events, traces and property, namely a vector and an iterator. Indeed, the main difficulty of our techniques in Sections 6 and 8 was precisely to handle general purpose parametric properties with an arbitrary number of parameters. The slicing algorithm in Section 6 processes parametric traces and maintains enough slicing information so that, when asked to produce slices corresponding to particular parameter instances, e.g., to "$v_1\, i_2$", it can do so without any further analysis of the trace. Also, the monitoring algorithm in Section 8 reports a match each time a parameter instance yields a matching trace slice.

3.3 Correct locking

Consider a custom implementation of synchronization in which one can acquire and release locks manually (like in Java 5). A basic property is that each function releases each lock as many times as it acquires it. Assuming that the executing code is always inside some function (like in Java, C, etc), that $\mathsf{begin}\langle\rangle$ and $\mathsf{end}\langle\rangle$ events are generated whenever function executions are started and terminated, respectively, and that $\mathsf{acquire}\langle l\rangle$ and $\mathsf{release}\langle l\rangle$ events are generated whenever lock l is acquired or released, respectively, then one can specify this safety property using the following parametric context-free grammar (CFG) pattern:

$$\Lambda l.S \to S\ \mathsf{begin}\ S\ \mathsf{end} \mid S\ \mathsf{acquire}\ S\ \mathsf{release} \mid \epsilon$$

Such parametric CFG properties can be expressed in both PQL [13] and JavaMOP [14] (the later using its CFG plugin). We here borrow the CFG property semantics of the CFG plugin of JavaMOP in [14], that is, this parametric property is violated by a parametric execution with a given parameter instance (i.e., concrete lock) whenever the corresponding trace slice cannot be completed into one accepted by the language of the grammar above. While this particular property can be expressed in JavaMOP and even monitored in its non-parametric form, the current implementation of JavaMOP in [14] cannot monitor it as a parametric property because its violating traces most likely start with a property-relevant $\mathsf{begin}\langle\rangle$ event, which does not contain a lock parameter; therefore, the current limitation of JavaMOP (allowing only events that instantiate all property's parameters to create a monitor instance) does not allow us to monitor this natural CFG property. To circumvent its limitation, JavaMOP proposes a different way to specify this property in [14], in which the violating traces start with an $\mathsf{acquire}\langle l\rangle$ event.

For profiling reasons, one may also want to take notice of validations, or matches of the property, as well as matches followed by violation, etc.; one can therefore have different interpretations of CFG patterns as base properties, classifying traces into various categories. What is different in this example, compared to the previous ones, is that the non-parametric property cannot be implemented as a finite state machine. With the CFG monitoring algorithm proposed in [14] used to monitor the base property, our parametric monitoring algorithm in Section 8 reports a violation of this parametric CFG property as soon as a parameter instance is detected for which the corresponding trace slice has no future, that is, it admits no continuation into a trace in the language of the grammar.

3.4 Safe resource use by safe client

A client can use a resource only within a given procedure and, when that happens, both the client and the resource must have been previously authenticated as part of that procedure. Assuming the procedure fixed and given, this is a property over traces with five types of events: begin and end of the procedure ($\mathsf{begin}\langle\rangle$ and $\mathsf{end}\langle\rangle$), authenticate of client ($\mathsf{auth\text{-}client}\langle c\rangle$) or of resource ($\mathsf{auth\text{-}resource}\langle r\rangle$), and use of resource by client ($\mathsf{use}\langle r,c\rangle$). Using the past time linear temporal logic with calls and returns (ptCaRet) in [16], one would write it:

$$\Lambda r,c.\;\; \mathsf{use} \to ((\overline{\diamond}\,\mathsf{begin}) \wedge \neg((\neg\mathsf{auth\text{-}client})\ \overline{\mathcal{S}}\ \mathsf{begin}) \\ \wedge \neg((\neg\mathsf{auth\text{-}resource})\ \overline{\mathcal{S}}\ \mathsf{begin})).$$

In ptCaRet, the overlined operators are abstract variants of temporal operators, abstract in the sense that they are defined on traces that collapse terminated procedure calls (erase subtraces bounded by matched begin/end events). For example, "$\overline{\diamond}\,\mathsf{begin}$" holds only within a procedure call, because all the nested and terminated procedure calls are abstracted away. In words, the above says: if one sees the use of the resource (use) then that must take place within the procedure and it is not the case that one did not see, within the main procedure since its latest invocation, the authentication of the client or the authentication of the resource.

JavaMOP can express this property using its ptCaRet logic plugin [16]. However, it can again only monitor it in its non-parametric form, because of its limitation allowing only completely parameterized events to create monitors. Even though it may appear that this property can only be violated when a completely parameterized $\mathsf{use}\langle r,c\rangle$ event is observed, in fact, the monitor must already exist at that point in the execution and "know" whether the client and the resource have authenticated since the begin of the current procedure; all the other events involved in the property are incompletely parameterized, so, unfortunately, this parametric property cannot be monitored by the current JavaMOP system [16].

3.5 Success ratio

Consider now parametric traces with events $\mathsf{success}\langle a\rangle$ and $\mathsf{fail}\langle a\rangle$, saying whether a certain action a was successful or not. For a given action, a meaningful property can classify its (non-parametric) traces into an infinite number of categories, each representing a success ratio of the given action, which is a (rational) number s/t between 0 and 1, where s is the number of success events in the trace and t is the total number of events in the trace. Then the corresponding parametric property over such parametric traces gives a success ratio for each action.

4. Least Upper Bound Closures of Partial Maps

In this section we first discuss some basic notions of partial functions and least upper bounds of them, then we introduce least upper bounds of sets of partial functions and least upper bound closures of sets of partial functions. This section is rather mathematical. We need these mathematical notions because, as already seen, parameter instances are partial maps from the domain of parameters to the domain of parameter values. As shown later, whenever a new parametric event is observed, it needs to be dispatched to the interested parts (trace slices or monitors), and those parts updated accordingly: these informal operations can be rigorously formalized as existence of least upper bounds and least upper bound closures over parameter instances, i.e., partial functions.

4.1 Partial Functions

We think of partial functions as "information carriers": if a partial function θ is defined on an element x of its domain, then "θ carries the information $\theta(x)$ about $x \in X$". Some partial functions can carry more information than others; two or more partial functions can, together, carry compatible information, but can also carry incompatible information (when two or more of them disagree on the information they carry for a particular $x \in X$). Recall that $[X \to V_X]$ and $[X \overset{\circ}{\to} V_X]$ represent the sets of *total* and of *partial functions* from X to V_X, respectively.

DEFINITION 6. *The domain of $\theta \in [X \overset{\circ}{\to} V_X]$ is the set $Dom(\theta) = \{x \in X \mid \theta(x) \text{ defined}\}$. Let $\bot \in [X \overset{\circ}{\to} V_X]$ be the map undefined everywhere, that is, $Dom(\bot) = \emptyset$. If $\theta, \theta' \in [X \overset{\circ}{\to} V_X]$ then we say that θ **is less informative than** θ', written $\theta \sqsubseteq \theta'$, if for any $x \in X$, $\theta(x)$ defined implies $\theta'(x)$ also defined and $\theta'(x) = \theta(x)$.*

It is known that $([X \overset{\circ}{\to} V_X], \sqsubseteq, \bot)$ is a complete (i.e., any $\sqsubseteq$-chain has a least upper bound) partial order with bottom (i.e., $\bot$).

DEFINITION 7. *Given $\Theta \subseteq [X \overset{\circ}{\to} V_X]$ and $\theta' \in [X \overset{\circ}{\to} V_X]$,*

- *θ' is an **upper bound** of Θ iff $\theta \sqsubseteq \theta'$ for any $\theta \in \Theta$; Θ **has upper bounds** iff there is a θ' which is an upper bound of Θ;*
- *θ' is the **least upper bound (lub)** of Θ iff θ' is an upper bound of Θ and $\theta' \sqsubseteq \theta''$ for any other upper bound θ'' of Θ;*
- *θ' is the **maximum (max)** of Θ iff $\theta' \in \Theta$ and θ' is a lub of Θ.*

Intuitively, a set of partial functions has an upper bound iff the partial functions in the set are *compatible*, that is, no two of them disagree on the value of a particular element in their domain. Least upper bounds and maximums may not always exist for any $\Theta \subseteq [X \overset{\circ}{\to} V_X]$; if a lub or a maximum for Θ exists, then it is, of course, unique ($\sqsubseteq$ is a partial order, so antisymmetric).

DEFINITION 8. *Given $\Theta \subseteq [X \overset{\circ}{\to} V_X]$, let $\sqcup\Theta$ and $\max\Theta$ be the lub and the max of Θ, respectively, when they exist. When Θ is finite, one may write $\theta_1 \sqcup \theta_2 \sqcup \cdots \sqcup \theta_n$ instead of $\sqcup\{\theta_1, \theta_2, \ldots, \theta_n\}$.*

If Θ has a maximum, then it also has a lub and $\sqcup\Theta = \max\Theta$. Here are several common properties that we use frequently:

PROPOSITION 1. *The following hold ($\theta, \theta_1, \theta_2, \theta_3 \in [X \overset{\circ}{\to} V_X]$): $\bot \sqcup \theta$ exists and $\bot \sqcup \theta = \theta$; $\theta_1 \sqcup \theta_2$ exists iff $\theta_2 \sqcup \theta_1$ exists, and, if they exist then $\theta_1 \sqcup \theta_2 = \theta_2 \sqcup \theta_1$; $\theta_1 \sqcup (\theta_2 \sqcup \theta_3)$ exists iff $(\theta_1 \sqcup \theta_2) \sqcup \theta_3$ exists, and if they exist then $\theta_1 \sqcup (\theta_2 \sqcup \theta_3) = (\theta_1 \sqcup \theta_2) \sqcup \theta_3$.*

PROPOSITION 2. *Let $\Theta \subseteq [X \overset{\circ}{\to} V_X]$. Then*

1. *Θ has an upper bound iff for any $\theta_1, \theta_2 \in \Theta$ and $x \in X$, if $\theta_1(x)$ and $\theta_2(x)$ defined then $\theta_1(x) = \theta_2(x)$;*
2. *If Θ has an upper bound then $\sqcup\Theta$ exists and, for any $x \in X$,*
$$(\sqcup\Theta)(x) = \begin{cases} \text{undefined} & \text{if } \theta(x) \text{ undefined for any } \theta \in \Theta \\ \theta(x) & \text{if there is a } \theta \in \Theta \text{ with } \theta(x) \text{ defined.} \end{cases}$$

Proof. Since Θ has an upper bound $\theta' \in [X \overset{\circ}{\to} V_X]$ iff $\theta \sqsubseteq \theta'$ for any $\theta \in \Theta$, if $\theta_1, \theta_2 \in \Theta$ and $x \in X$ with $\theta_1(x)$ and $\theta_2(x)$ defined then $\theta'(x)$ is also defined and $\theta_1(x) = \theta_2(x) = \theta'(x)$. Suppose now that for any $\theta_1, \theta_2 \in \Theta$ and $x \in X$, if $\theta_1(x)$ and $\theta_2(x)$ defined then $\theta_1(x) = \theta_2(x)$. All we need to show in order to prove both results is that we can find a lub for Θ. Let $\theta' \in [X \overset{\circ}{\to} V_X]$ be defined as follows: for any $x \in X$, let

$$\theta'(x) = \begin{cases} \text{undefined} & \text{if } \theta(x) \text{ undefined for any } \theta \in \Theta \\ \theta(x) & \text{if there is a } \theta \in \Theta \text{ such that } \theta(x) \text{ defined} \end{cases}$$

First, note that θ' above is indeed well-defined, because we assumed that for any $\theta_1, \theta_2 \in \Theta$ and $x \in X$, if $\theta_1(x)$ and $\theta_2(x)$ defined then $\theta_1(x) = \theta_2(x)$. Second, note that θ' is an upper bound for Θ: indeed, if $\theta \in \Theta$ and $x \in X$ such that $\theta(x)$ defined, then $\theta'(x)$ is also defined and $\theta'(x) = \theta(x)$, that is, $\theta \sqsubseteq \theta'$ for any $\theta \in \Theta$. Finally, θ' is a lub for Θ: if θ'' is another upper bound for Θ and $\theta'(x)$ defined for some $x \in X$, that is, $\theta(x)$ defined for some $\theta \in \Theta$ and $\theta'(x) = \theta(x)$, then $\theta''(x)$ also defined and $\theta'(x) = \theta(x)$ (as $\theta \sqsubseteq \theta''$), so $\theta' \sqsubseteq \theta''$. □

PROPOSITION 3. *The following hold:*

1. *The empty set of partial functions $\emptyset \subseteq [X \overset{\circ}{\to} V_X]$ has upper bounds and $\sqcup\emptyset = \bot$;*
2. *The one-element sets have upper bounds and $\sqcup\{\theta\} = \theta$ for any $\theta \in [X \overset{\circ}{\to} V_X]$;*
3. *The bottom "$\bot$" does not influence the least upper bounds: $\sqcup(\{\bot\} \cup \Theta) = \sqcup\Theta$ for any $\Theta \subseteq [X \overset{\circ}{\to} V_X]$;*
4. *If $\Theta, \Theta' \subseteq [X \overset{\circ}{\to} V_X]$ s.t. $\sqcup\Theta'$ exists and for any $\theta \in \Theta$ there is a $\theta' \in \Theta'$ with $\theta \sqsubseteq \theta'$, then $\sqcup\Theta$ exists and $\sqcup\Theta \sqsubseteq \sqcup\Theta'$; e.g., if $\sqcup\Theta'$ exists and $\Theta \subseteq \Theta'$ then $\sqcup\Theta$ exists and $\sqcup\Theta \sqsubseteq \sqcup\Theta'$;*
5. *Let $\{\Theta_i\}_{i \in I}$ be a family of sets of partial functions with $\Theta_i \subseteq [X \overset{\circ}{\to} V_X]$. Then $\sqcup \cup \{\Theta_i \mid i \in I\}$ exists iff $\sqcup\{\sqcup\Theta_i \mid i \in I\}$ exists, and, if both exist, $\sqcup \cup \{\Theta_i \mid i \in I\} = \sqcup\{\sqcup\Theta_i \mid i \in I\}$.*

Proof. *1.*, *2.* and *3.* are straightforward. For *4.*, since for each $\theta \in \Theta$ there is some $\theta' \in \Theta'$ with $\theta \sqsubseteq \theta'$, and since $\theta' \sqsubseteq \sqcup\Theta'$ for any $\theta' \in \Theta'$, it follows that $\theta \sqsubseteq \sqcup\Theta'$ for any $\theta \in \Theta$, that is, that $\sqcup\Theta'$ is an upper bound for Θ. Therefore, by Proposition 2 it follows that $\sqcup\Theta$ exists and $\sqcup\Theta \sqsubseteq \sqcup\Theta'$ (the latter because $\sqcup\Theta$ is the *least* upper bound of Θ). We prove *5.* by double implication, each implication stating that if one of the lub's exist then the other one also exists and one of the inclusions holds; that indeed implies that one of the lub's exists if and only if the other one exists and, if both exist, then they are equal. Suppose first that $\sqcup \cup \{\Theta_i \mid i \in I\}$ exists, that is, that $\cup\{\Theta_i \mid i \in I\}$ has an upper bound, say u. Since $\Theta_i \subseteq \cup\{\Theta_i \mid i \in I\}$ for each $i \in I$, it follows first that each Θ_i also has u as an upper bound, so all $\sqcup\Theta_i$ for all $i \in I$ exist, and second by *4.* above that $\sqcup\Theta_i \sqsubseteq \sqcup \cup \{\Theta_i \mid i \in I\}$ for each $i \in I$. Item *4.* above then further implies that $\sqcup\{\sqcup\Theta_i \mid i \in I\}$ exists and $\sqcup\{\sqcup\Theta_i \mid i \in I\} \sqsubseteq \sqcup\{\sqcup \cup \{\Theta_i \mid i \in I\}\} = \sqcup \cup \{\Theta_i \mid i \in I\}$ (the last equality follows by *2.* above). Conversely, suppose now that $\sqcup\{\sqcup\Theta_i \mid i \in I\}$ exists. Since for each $\theta \in \cup\{\Theta_i \mid i \in I\}$ there is some $i \in I$ such that $\theta \sqsubseteq \sqcup\Theta_i$ (an $i \in I$ such that $\theta \in \Theta_i$), item *4.* above implies that $\sqcup \cup \{\Theta_i \mid i \in I\}$ also exists and $\sqcup \cup \{\Theta_i \mid i \in I\} \sqsubseteq \sqcup\{\sqcup\Theta_i \mid i \in I\}$. □

4.2 Least Upper Bounds of Families of Sets of Partial Maps

The notions of partial function, upper bound and least upper bound above are broadly known, and many of their properties are folklore. Motivated by requirements and optimizations of our trace slicing and monitoring algorithms in Sections 6 and 8, we next define several less known notions. We are actually not aware of other places where these notions are defined, so they could be novel.

We first extend the notion of lub from one associating a partial function to a set of partial functions to one associating a set of partial functions to a family (or set) of sets of partial functions:

DEFINITION 9. *If $\{\Theta_i\}_{i \in I}$ is a family of sets in $[X \overset{\circ}{\to} V_X]$, then we let the **least upper bound** (also **lub**) of $\{\Theta_i\}_{i \in I}$ be defined as:*

$$\sqcup\{\Theta_i \mid i \in I\} \overset{\text{def}}{=} \{\sqcup\{\theta_i \mid i \in I\} \mid \theta_i \in \Theta_i \textit{ for each } i \in I \textit{ s.t. } \sqcup\{\theta_i \mid i \in I\} \textit{ exists}\}.$$

As before, we use the infix notation when I is finite, e.g., we may write $\Theta_1 \sqcup \Theta_2 \sqcup \cdots \sqcup \Theta_n$ instead of $\sqcup\{\Theta_i \mid i \in \{1, 2, \ldots, n\}\}$.

Therefore, $\sqcup\{\Theta_i \mid i \in I\}$ is the set containing all the lub's corresponding to sets formed by picking for each $i \in I$ precisely one element from Θ_i. Unlike for sets of partial functions, the lub's of families of sets of partial functions always exist; $\sqcup\{\Theta_i \mid i \in I\}$ is the empty set when no collection of $\theta_i \in \Theta_i$ can be found (one $\theta_i \in \Theta_i$ for each $i \in I$) such that $\{\theta_i \mid i \in I\}$ has an upper bound.

There is an admitted slight notational ambiguity between the two least upper bound notations introduced so far. We prefer to purposely allow this ambiguity instead of inventing a new notation for the lub's of families of sets, hoping that the reader is able to quickly disambiguate the two by checking the types of the objects involved in the lub: if partial functions then the first lub is meant, if sets of partial functions then the second. Note that such notational ambiguities are actually common practice elsewhere; e.g., in a monoid $(M, \_*\_ : M \times M \to M, 1)$ with binary operation $*$ and unit 1, the $*$ is commonly extended to sets of elements M_1, M_2 in M as expected: $M_1 * M_2 = \{m_1 * m_2 \mid m_1 \in M_1, m_2 \in M_2\}$.

PROPOSITION 4. *The next hold ($\Theta, \Theta_1, \Theta_2, \Theta_3 \subseteq [X \overset{\circ}{\to} V_X]$):*

1. *$\sqcup\emptyset = \{\bot\}$, where, in this case, $\emptyset \subseteq \mathcal{P}([X \overset{\circ}{\to} V_X])$;*
2. *$\sqcup\{\Theta\} = \Theta$; in particular $\sqcup\{\emptyset\} = \emptyset$, where $\emptyset \subseteq [X \overset{\circ}{\to} V_X]$;*
3. *$\sqcup\{\{\theta\} \mid \theta \in \Theta\} = \begin{cases} \{\sqcup\Theta\} & \textit{if } \Theta \textit{ has a lub, and} \\ \emptyset & \textit{if } \Theta \textit{ does not have a lub;} \end{cases}$*
4. *$\emptyset \sqcup \Theta = \emptyset$, where $\emptyset \subseteq [X \overset{\circ}{\to} V_X]$;*
5. *$\{\bot\} \sqcup \Theta = \Theta$;*
6. *If $\Theta_1 \subseteq \Theta_2$ then $\Theta_1 \sqcup \Theta_3 \subseteq \Theta_2 \sqcup \Theta_3$; in particular, if $\bot \in \Theta_2$ then $\Theta_3 \subseteq \Theta_2 \sqcup \Theta_3$;*
7. *$(\Theta_1 \cup \Theta_2) \sqcup \Theta_3 = (\Theta_1 \sqcup \Theta_3) \cup (\Theta_2 \sqcup \Theta_3)$.*

Proof. Recall that the least upper bound $\sqcup\{\Theta_i \mid i \in I\}$ of sets of sets of partial functions is built by collecting all the least upper bounds of sets $\{\theta_i \mid i \in I\}$ containing one element θ_i from each of the involved sets Θ_i. When $|I| = 0$, that is when I is empty, there is precisely one set $\{\theta_i \mid i \in I\}$, the empty set of partial functions. Then *1.* follows by *1.* in Proposition 3. When $|I| = 1$, that is when $\{\Theta_i \mid i \in I\} = \{\Theta\}$ for some $\Theta \subseteq [X \overset{\circ}{\to} V_X]$ like in *2.*, then the sets $\{\theta_i \mid i \in I\}$ are precisely the singleton sets corresponding to the elements of Θ, so *2.* follows by *2.* in Proposition 3. *3.* holds because there is only one way to pick an element from each singleton set $\{\theta\}$, namely to pick the θ itself; this also shows how the notion of a lub of a family of sets generalizes the conventional notion of lub. When $|I| \geq 2$ and at least one of the involved sets of partial functions is empty, like in *4.*, then there is no set $\{\theta_i \mid i \in I\}$, so the least upper bound of the set of sets is empty (regarded, again, as the empty set of sets of partial functions). *5.* follows by *1.* in Proposition 1. The first part of *6.* is immediate and the second part follows from the first using *5.*. Finally, *7.* follows by double implication: $(\Theta_1 \sqcup \Theta_3) \cup (\Theta_2 \sqcup \Theta_3) \subseteq (\Theta_1 \cup \Theta_2) \sqcup \Theta_3$ follows by *6.* because Θ_1 and Θ_2 are included in $\Theta_1 \cup \Theta_2$, and $(\Theta_1 \cup \Theta_2) \sqcup \Theta_3 \subseteq (\Theta_1 \sqcup \Theta_3) \cup (\Theta_2 \sqcup \Theta_3)$ because for any $\theta_1 \in \Theta_1 \cup \Theta_2$, say $\theta_1 \in \Theta_1$, and any $\theta_3 \in \Theta_3$, if $\theta_1 \sqcup \theta_3$ exists then it also belongs to $\Theta_1 \sqcup \Theta_3$. □

PROPOSITION 5. *Let $\{\Theta_i\}_{i \in I}$ be a family of sets of partial maps in $[X \overset{\circ}{\to} V_X]$ and let $\mathcal{I} = \{I_j\}_{j \in J}$ be a partition of I: $I = \cup\{I_j \mid j \in J\}$ and $I_{j_1} \cap I_{j_2} = \emptyset$ for any different $j_1, j_2 \in J$. Then*

$$\sqcup\{\Theta_i \mid i \in I\} = \sqcup\{\sqcup\{\Theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}.$$

Proof. For each $j \in J$, let Q_j denote the set $\sqcup\{\Theta_{i_j} \mid i_j \in I_j\}$. Definition 9 then implies the following: $Q_j \overset{\text{def}}{=} \{\sqcup\{\theta_{i_j} \mid i_j \in I_j\} \mid \theta_{i_j} \in \Theta_{i_j}$ for each $i_j \in I_j$, such that $\sqcup\{\theta_{i_j} \mid i_j \in I_j\}$ exists$\}$. Definition 9 also implies the following: $\sqcup\{Q_j \mid j \in J\} \overset{\text{def}}{=} \{\sqcup\{q_j \mid j \in J\} \mid q_j \in Q_j$ for each $j \in J$, such that $\sqcup\{q_j \mid j \in J\}$ exists$\}$. Putting the two equalities above together, we get that $\sqcup\{\sqcup\{\Theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}$ equals the following:

$$\begin{aligned}\{\ \sqcup\{\sqcup\{\theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}\\ \mid \theta_{i_j} \in \Theta_{i_j} \text{ for each } j \in J \text{ and } i_j \in I_j, \text{ such that}\\ \sqcup\{\theta_{i_j} \mid i_j \in I_j\} \text{ exists for each } j \in J \text{ and}\\ \sqcup\{\sqcup\{\theta_{i_j} \mid i_j \in I_j\} \mid j \in J\} \text{ exists}\}.\end{aligned}$$

Since $\{I_j\}_{j\in J}$ is a partition of I, the indices i_j generated by "for each $j \in J$ and $i_j \in I_j$" cover precisely all the indices $i \in I$. Moreover, picking partial functions $\theta_{i_j} \in \Theta_{i_j}$ for each $j \in J$ and $i_j \in I_j$ is equivalent to picking partial functions $\theta_i \in \Theta_i$ for each $i \in I$, and, in this case, $\{\theta_i \mid i \in I\} = \cup\{\{\theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}$. By *5.* in Proposition 3 we then infer that $\sqcup\{\theta_i \mid i \in I\}$ exists if and only if $\sqcup\{\sqcup\{\theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}$ exists, and if both exist then $\sqcup\{\theta_i \mid i \in I\} = \sqcup\{\sqcup\{\theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}$; if both exist then $\sqcup\{\theta_{i_j} \mid i_j \in I_j\}$ also exists for each $j \in J$ (because $\{\theta_{i_j} \mid i_j \in I_j\} \subseteq \{\theta_i \mid i \in I\} = \cup\{\{\theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}$). Therefore, we can conclude that $\sqcup\{\sqcup\{\Theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}$ equals $\{\sqcup\{\theta_i \mid i \in I\} \mid \theta_i \in \Theta_i$ for each $i \in I$, such that $\sqcup\{\theta_i \mid i \in I\}$ exists$\}$, which is nothing but $\sqcup\{\Theta_i \mid i \in I\}$. □

COROLLARY 1. *The following hold:*

1. $\{\bot\} \sqcup \Theta = \Theta$ *(already proved as 5. in Proposition 4);*
2. $\Theta_1 \sqcup \Theta_2 = \Theta_2 \sqcup \Theta_1$*;*
3. $\Theta_1 \sqcup (\Theta_2 \sqcup \Theta_3) = (\Theta_1 \sqcup \Theta_2) \sqcup \Theta_3$*;*

Proof. These follow from Proposition 5 for various index sets I and partitions of it: for *1.* take $I = \{1\}$ and its partition $I = \emptyset \cup I$, take $\Theta_1 = \Theta$, and then use *1.* in Proposition 4 saying that $\sqcup\emptyset = \{\bot\}$; for *2.* take partitions $\{1\}\cup\{2\}$ and $\{2\}\cup\{1\}$ of $I = \{1, 2\}$, getting $\Theta_1 \sqcup \Theta_2 = \Theta_2 \sqcup \Theta_1 = \sqcup\{\Theta_i \mid i \in \{1, 2\}\}$; finally, for *3.* take partitions $\{1\} \cup \{2, 3\}$ and $\{1, 2\} \cup \{3\}$ of $I = \{1, 2, 3\}$, getting $\Theta_1 \sqcup (\Theta_2 \sqcup \Theta_3) = (\Theta_1 \sqcup \Theta_2) \sqcup \Theta_3 = \sqcup\{\Theta_i \mid i \in \{1, 2, 3\}\}$. □

4.3 Least Upper Bound Closures

We next define lub closures of sets of partial maps, a crucial operation for the the algorithms discussed next in the paper.

DEFINITION 10. $\Theta \subseteq [X \overset{\circ}{\to} V_X]$ *is **lub closed** iff* $\sqcup\Theta' \in \Theta$ *for any* $\Theta' \subseteq \Theta$ *admitting upper bounds.*

PROPOSITION 6. $\{\bot\}$ *and* $\{\bot, \theta\}$ *are lub closed (*$\theta \in [X \overset{\circ}{\to} V_X]$*).*

Proof. It follows easily from Definition 10, using the facts that $\sqcup\emptyset = \bot$ (*1.* in Proposition 3), $\sqcup\{\theta\} = \theta$ (*2.* in Proposition 3), and $\sqcup\{\bot, \theta\} = \theta$ (*3.* in Proposition 3 for $\Theta = \{\theta\}$). □

PROPOSITION 7. *If* $\Theta \subseteq [X \overset{\circ}{\to} V_X]$ *and* $\{\Theta_i\}_{i\in I}$ *is a family of sets of partial functions in* $[X \overset{\circ}{\to} V_X]$*, then:*

1. *If* Θ *is lub closed then* $\bot \in \Theta$*; in particular,* $\emptyset$ *is not lub closed;*
2. *If* Θ *has upper bounds and is lub closed then it has a maximum;*
3. Θ *is lub closed iff* $\sqcup\{\Theta \mid i \in I\} = \Theta$ *for any* I*;*
4. *If* Θ *is lub closed and* $\Theta_i \subseteq \Theta$ *for each* $i \in I$ *then* $\sqcup\{\Theta_i \mid i \in I\} \subseteq \Theta$*;*
5. *If* Θ_i *is lub closed for each* $i \in I$ *then* $\sqcup\{\Theta_i \mid i \in I\}$ *is lub closed and* $\cup\{\Theta_i \mid i \in I\} \subseteq \sqcup\{\Theta_i \mid i \in I\}$*;*
6. *If* I *finite and* Θ_i *finite for all* $i \in I$*, then* $\sqcup\{\Theta_i \mid i \in I\}$ *finite;*
7. *If* Θ_i *lub closed for all* $i \in I$ *then* $\cap\{\Theta_i \mid i \in I\}$ *is lub closed;*
8. $\cap\{\Theta' \mid \Theta' \subseteq [X \overset{\circ}{\to} V_X]$ *with* $\Theta \subseteq \Theta'$ *and* Θ' *is lub closed*$\}$ *is the smallest lub closed set including* Θ*.*

Proof. *1.* follows taking $\Theta' = \emptyset$ in Definition 10 and using $\sqcup\emptyset = \bot$ (*1.* in Proposition 3).

2. follows taking $\Theta' = \Theta$ in Definition 10: $\sqcup\Theta \in \Theta$, so $\max\Theta$ exists (and equals $\sqcup\Theta$).

3. Definition 9 implies that $\sqcup\{\Theta \mid i \in I\}$ equals $\{\sqcup\{\theta_i \mid i \in I\} \mid \theta_i \in \Theta$ for each $i \in I$, such that $\sqcup\{\theta_i \mid i \in I\}$ exists$\}$, which is nothing but $\{\sqcup\Theta' \mid \Theta' \subseteq \Theta$ such that $\sqcup\Theta'$ exists$\}$; the later can be now shown equal to Θ by double inclusion: $\{\sqcup\Theta' \mid \Theta' \subseteq \Theta$ such that $\sqcup\Theta'$ exists$\} \subseteq \Theta$ because Θ is lub closed, and $\Theta \subseteq \{\sqcup\Theta' \mid \Theta' \subseteq \Theta$ such that $\sqcup\Theta'$ exists$\}$ because one can pick $\Theta' = \{\theta\}$ for each $\theta \in \Theta$ and use the fact that $\sqcup\{\theta\} = \theta$ (*2.* in Proposition 3).

4. Let θ be an arbitrary partial function in $\sqcup\{\Theta_i \mid i \in I\}$, that is, $\theta = \sqcup\{\theta_i \mid i \in I\}$ for some $\theta_i \in \Theta_i$, one for each $i \in I$, such that $\{\theta_i \mid i \in I\}$ has upper bounds. Since Θ is lub closed and $\Theta_i \subseteq \Theta$ for each $i \in I$, it follows that $\theta \in \Theta$. Therefore, $\sqcup\{\Theta_i \mid i \in I\} \subseteq \Theta$.

5. Let Θ' be a set of partial functions included in $\sqcup\{\Theta_i \mid i \in I\}$ which admits an upper bound; moreover, for each $\theta' \in \Theta'$, let us fix a set $\{\theta_i^{\theta'} \mid i \in I\}$ such that $\theta_i^{\theta'} \in \Theta_i$ for each $i \in I$ and $\theta' = \sqcup\{\theta_i^{\theta'} \mid i \in I\}$ (such sets exist because $\theta' \in \Theta' \subseteq \sqcup\{\Theta_i \mid i \in I\}$). Let $\Theta^{\theta'}$ be the set $\{\theta_i^{\theta'} \mid i \in I\}$ for each $\theta' \in \Theta'$, let Θ'_i be the set $\{\theta_i^{\theta'} \mid \theta' \in \Theta'\}$ for each $i \in I$, and let Θ be the set $\{\theta_i^{\theta'} \mid \theta' \in \Theta', i \in I^{\theta'}\}$. It is easy to see that $\Theta = \cup\{\Theta^{\theta'} \mid \theta' \in \Theta'\} = \cup\{\Theta'_i \mid i \in I\}$ and that $\Theta'_i \subseteq \Theta_i$ for each $i \in I$. Since $\sqcup\Theta'$ exists (because Θ' has upper bounds) and $\sqcup\Theta' = \sqcup\{\theta' \mid \theta' \in \Theta'\} = \sqcup\{\sqcup\Theta^{\theta'} \mid \theta' \in \Theta'\}$, by *5.* in Proposition 3 it follows that $\sqcup\Theta$ exists and $\sqcup\Theta' = \sqcup\Theta$. Since $\Theta = \cup\{\Theta'_i \mid i \in I\}$ and $\sqcup\Theta$ exists, by *5.* in Proposition 3 again we get that $\sqcup\{\sqcup\Theta'_i \mid i \in I\}$ exists and is equal to $\sqcup\Theta$, which is equal to $\sqcup\Theta'$. Since $\Theta'_i \subseteq \Theta_i$ and Θ_i is lub closed, we get that $\sqcup\Theta'_i \in \Theta_i$. That means that $\sqcup\{\sqcup\Theta'_i \mid i \in I\} \in \sqcup\{\Theta_i \mid i \in I\}$, that is, that $\sqcup\Theta' \in \sqcup\{\Theta_i \mid i \in I\}$. Since $\Theta' \subseteq \sqcup\{\Theta_i \mid i \in I\}$ was chosen arbitrarily, we conclude that $\sqcup\{\Theta_i \mid i \in I\}$ is lub closed. To show that $\cup\{\Theta_i \mid i \in I\} \subseteq \sqcup\{\Theta_i \mid i \in I\}$, let us pick an $i \in I$ and let us partition I as $\{i\} \cup (I\backslash\{i\})$. By Proposition 5, $\sqcup\{\Theta_i \mid i \in I\} = \Theta_i \sqcup (\sqcup\{\Theta_j \mid j \in I\backslash\{i\}\})$. The proof above also implies that $\sqcup\{\Theta_j \mid j \in I\backslash\{i\}\}$ is lub closed, so by *1.* we get that $\bot \in \sqcup\{\Theta_j \mid j \in I\backslash\{i\}\}$. Finally, *6.* in Proposition 4 implies $\Theta_i \sqsubseteq \Theta_i \sqcup (\sqcup\{\Theta_j \mid j \in I\backslash\{i\}\})$, so $\Theta_i \subseteq \sqcup\{\Theta_i \mid i \in I\}$ for each $i \in I$, that is, $\cup\{\Theta_i \mid i \in I\} \subseteq \sqcup\{\Theta_i \mid i \in I\}$.

6. Recall from Definition 9 that $\sqcup\{\Theta_i \mid i \in I\}$ contains the existing least upper bounds of sets of partial functions containing precisely one partial function in each Θ_i. If I and each of the Θ_i for each $i \in I$ is finite, then $|\sqcup\{\Theta_i \mid i \in I\}| \leq \prod_{i\in I} |\Theta_i|$, because there at most $\prod_{i\in I} |\Theta_i|$ combinations of partial functions, one in each Θ_i, that admit an upper bound. Therefore, $\sqcup\{\Theta_i \mid i \in I\}$ is also finite.

7. Let $\Theta' \subseteq \cap\{\Theta_i \mid i \in I\}$ be a set of partial functions admitting an upper bound. Then $\Theta' \subseteq \Theta_i$ for each $i \in I$ and, since each Θ_i is lub closed, $\sqcup\Theta' \in \Theta_i$ for each $i \in I$. Therefore, $\sqcup\Theta' \in \cap\{\Theta_i \mid i \in I\}$.

8. Anticipating the definition of and notation for lub closures (Definition 10), we let $\overline{\Theta}$ denote the set $\cap\{\Theta' \mid \Theta' \subseteq [X \overset{\circ}{\to} V_X]$ with $\Theta \subseteq \Theta'$ and Θ' is lub closed$\}$. It is clear that $\Theta \subseteq \overline{\Theta}$ and, by *7.*, that $\overline{\Theta}$ is lub closed. It is also the *smallest* lub closed set including Θ, because all such sets Θ' are among those whose intersection defines $\overline{\Theta}$. □

DEFINITION 11. *Given $\theta' \in [X \overset{\circ}{\to} V_X]$ and $\Theta \subseteq [X \overset{\circ}{\to} V_X]$, let*

$$(\theta']_\Theta \overset{\text{def}}{=} \{\theta \mid \theta \in \Theta \text{ and } \theta \sqsubseteq \theta'\}$$

be the set of partial functions in Θ that are less informative than θ'.

PROPOSITION 8. *If $\theta, \theta', \theta'', \theta_1, \theta_2 \in [X \overset{\circ}{\to} V_X]$ and if $\Theta \subseteq [X \overset{\circ}{\to} V_X]$ is lub closed, then:*

1. *$(\theta']_\Theta$ is lub closed and $\max(\theta']_\Theta$ exists;*
2. *If $\theta' \in \{\theta\} \sqcup \Theta$ then $\{\theta'' \mid \theta'' \in \Theta$ and $\theta' = \theta \sqcup \theta''\}$ has maximum and that equals $\max(\theta']_\Theta$;*
3. *If $\theta_1, \theta_2 \in \{\theta\} \sqcup \Theta$ such that $\theta_1 = \max(\theta_2]_\Theta$, then $\theta_1 = \theta_2$.*

Proof. *1.* First, note that θ' is an upper bound for $(\theta']_\Theta$ as well as for any subset Θ' of it, so any $\Theta' \subseteq (\theta']_\Theta$ has upper bounds, so by *2.* in Proposition 2, $\sqcup\Theta'$ exists for any $\Theta' \subseteq (\theta']_\Theta$. Moreover, if $\Theta' \subseteq (\theta']_\Theta$ then $\sqcup\Theta' \sqsubseteq \theta'$, and since Θ is lub closed it follows that $\sqcup\Theta' \in \Theta$, so $\sqcup\Theta' \in (\theta']_\Theta$. Therefore, $(\theta']_\Theta$ is lub closed. *2.* in Proposition 7 now implies that $(\theta']_\Theta$ has maximum; to be concrete, $\max(\theta']_\Theta$ is nothing but $\sqcup(\theta']_\Theta$, which belongs to $(\theta']_\Theta$ (because one can pick $\Theta' = (\theta']_\Theta$ above).

2. Let Q be the set of partial functions $\{\theta'' \mid \theta'' \in \Theta$ and $\theta' = \theta \sqcup \theta''\}$. Note that Q is non-empty (because $\theta' \in \{\theta\} \sqcup \Theta$, so there is some $\theta'' \in \Theta$ such as $\theta' = \theta \sqcup \theta''$) and has upper-bounds (because θ' is an upper bound for it), but that it is not necessarily lub closed (because, unless $\theta' = \theta$, Q does not contain $\perp$, contradicting *1.* in Proposition 7). Hence Q has a lub (by *2.* in Proposition 2), say q, and $q = \sqcup Q \sqsubseteq \theta'$; since $\theta \sqsubseteq \theta'$, it follows that $\theta \sqcup q \sqsubseteq \theta'$. On the other hand $\theta' \sqsubseteq \theta \sqcup q$ by *4.* in Proposition 3, because there is some $\theta'' \in Q$ such that $\theta' = \theta \sqcup \theta''$ and $\theta'' \sqsubseteq q$. Therefore, $\theta' = \theta \sqcup q$. Since Θ is lub closed, it follows that $q \in \Theta$. Therefore, $q \in Q$, so q is the maximum element of Q. Let us next show that $q = \max(\theta']_\Theta$. The relation $q \sqsubseteq \max(\theta']_\Theta$ is immediate because $q \in (\theta']_\Theta$ (we proved above that $q \in \Theta$ and $q \sqsubseteq \theta'$). For $\max(\theta']_\Theta \sqsubseteq q$ it suffices to show that $\max(\theta']_\Theta \in Q$, that is, that $\theta \sqcup \max(\theta']_\Theta = \theta'$: $\theta \sqcup \max(\theta']_\Theta \sqsubseteq \theta'$ follows because $\theta \sqsubseteq \theta'$ and $\max(\theta']_\Theta \sqsubseteq \theta'$, while $\theta' \sqsubseteq \theta \sqcup \max(\theta']_\Theta$ follows because there is some $\theta'' \in \Theta$ such that $\theta' = \theta \sqcup \theta''$ and, since $\theta'' \sqsubseteq \max(\theta']_\Theta$, $\theta \sqcup \theta'' \sqsubseteq \theta \sqcup \max(\theta']_\Theta$ (by *4.* in Proposition 3).

3. admits a direct proof simpler than that of *2.*; however, since *2.* is needed anyway, we prefer to use *2.* Note that $\theta \sqsubseteq \theta_1 \sqsubseteq \theta_2$. By *2.*, $\theta_1 = \max\{\theta'' \mid \theta'' \in \Theta$ and $\theta_2 = \theta \sqcup \theta''\}$, which implies $\theta_2 = \theta \sqcup \theta_1 = \theta_1$. □

DEFINITION 12. *Given $\Theta \subseteq [X \overset{\circ}{\to} V_X]$, we let $\overline{\Theta}$, the* ***least upper bound (lub) closure*** *of Θ, be defined as follows:*

$$\overline{\Theta} \overset{\text{def}}{=} \cap\{\Theta' \mid \Theta' \subseteq [X \overset{\circ}{\to} V_X] \text{ with } \Theta \subseteq \Theta' \text{ and } \Theta' \text{ is lub closed}\}.$$

PROPOSITION 9. *The next hold ($\Theta \subseteq [X \overset{\circ}{\to} V_X], \theta \in [X \overset{\circ}{\to} V_X]$):*

1. *$\overline{\Theta}$ is the smallest lub closed set including Θ;*
2. *$\overline{\emptyset} = \overline{\{\perp\}} = \{\perp\}$;*
3. *$\overline{\{\theta\}} = \{\perp, \theta\}$.*

Proof. *1.* follows by *7.* in Proposition 7. For *2.* and *3.*, first note that $\{\perp\}$ and $\{\perp, \theta\}$ are lub closed by Proposition 6; second, note that they are indeed the smallest lub closed sets including $\perp$ and resp. θ, as any lub closed set must include $\perp$ (*1.* in Proposition 7). □

PROPOSITION 10. *The lub closure map $\overline{\cdot} : 2^{[X \overset{\circ}{\to} V_X]} \to 2^{[X \overset{\circ}{\to} V_X]}$ is a closure operator, that is, for any $\Theta, \Theta_1, \Theta_2 \subseteq [X \overset{\circ}{\to} V_X]$,*

1. *(extensivity) $\Theta \subseteq \overline{\Theta}$;*
2. *(monotonicity) If $\Theta_1 \subseteq \Theta_2$ then $\overline{\Theta_1} \subseteq \overline{\Theta_2}$;*
3. *(idempotency) $\overline{\overline{\Theta}} = \overline{\Theta}$.*

Proof. Extensivity and idempotency follow immediately from the definitions of $\overline{\Theta}$ and $\overline{\overline{\Theta}}$ (which are lub closed by *1.* in Proposition 9). For monotonicity, one should note that $\overline{\Theta_2}$ satisfies the properties of $\overline{\Theta_1}$ (i.e., $\Theta_1 \subseteq \overline{\Theta_2}$ and Θ_2 is lub closed); since $\overline{\Theta_1}$ is the smallest with those properties, it follows that $\overline{\Theta_1} \subseteq \overline{\Theta_2}$. □

PROPOSITION 11. *$\overline{\cup\{\Theta_i \mid i \in I\}} = \overline{\sqcup\{\overline{\Theta_i} \mid i \in I\}}$ for any family $\{\Theta_i\}_{i \in I}$ of partial functions in $[X \overset{\circ}{\to} V_X]$.*

Proof. Since $\overline{\Theta_i}$ is lub closed for any $i \in I$, *5.* in Proposition 7 implies that $\sqcup\{\overline{\Theta_i} \mid i \in I\}$ is lub closed and $\cup\{\overline{\Theta_i} \mid i \in I\} \subseteq \sqcup\{\overline{\Theta_i} \mid i \in I\}$. Since *1.* in Proposition 10 implies $\Theta_i \subseteq \overline{\Theta_i}$ for each $i \in I$ and since $\overline{\cup\{\Theta_i \mid i \in I\}}$ is the smallest lub closed set including $\cup\{\Theta_i \mid i \in I\}$ (*1.* in Proposition 9), the inclusion $\overline{\cup\{\Theta_i \mid i \in I\}} \subseteq \sqcup\{\overline{\Theta_i} \mid i \in I\}$ holds. Conversely, *2.* in Proposition 10 implies that $\overline{\Theta_i} \subseteq \overline{\cup\{\Theta_i \mid i \in I\}}$ for any $i \in I$, so $\sqcup\{\overline{\Theta_i} \mid i \in I\} \subseteq \overline{\cup\{\Theta_i \mid i \in I\}}$ holds by *4.* in Proposition 7. □

COROLLARY 2. *For any $\theta \in [X \overset{\circ}{\to} V_X]$ and any $\Theta \subseteq [X \overset{\circ}{\to} V_X]$, the equality $\overline{\{\theta\} \cup \Theta} = \{\perp, \theta\} \sqcup \overline{\Theta}$ holds.*

Proof. $\overline{\{\theta\} \cup \Theta} = \overline{\{\theta\}} \sqcup \overline{\Theta}$ by Proposition 11, and further $\overline{\{\theta\}} = \{\perp, \theta\}$ by *3.* in Proposition 9. □

COROLLARY 3. *If $\Theta \subseteq [X \overset{\circ}{\to} V_X]$ is finite then $\overline{\Theta}$ is also finite.*

Proof. Suppose that $\Theta = \{\theta_1, \theta_2, \ldots, \theta_n\}$ for some $n \geq 0$. Iteratively applying Corollary 2, $\overline{\Theta} = \{\perp, \theta_1\} \sqcup \{\perp, \theta_2\} \sqcup \cdots \{\perp, \theta_n\}$; in obtaining that, we used *2.* in Proposition 9 and *1.* in Corollary 1. The result follows now by *6.* in Proposition 7. □

5. Slicing With Less

Consider a parametric trace τ in $\mathcal{E}\langle X\rangle^*$ and a parameter instance θ. Since there is no apriori correlation between the parameters being instantiated by θ and those by the various parametric events in τ, it may very well be the case that θ contains parameter instances that never appear in τ. In this section we show that slicing τ by θ is the same as slicing it by a "smaller" parameter instance than θ, namely one containing only those parameters instantiated by θ that also appear as instances of some parameters in some events in τ. Formally, this smaller parameter instance is the largest partial map smaller than θ in the lub closure of all the parameter instances of events in τ; this partial function is proved to indeed exist. We first formalize a notation used informally so far in this paper:

NOTATION 1. *When the domain of θ is finite, which is always the case in our examples in this paper and probably will also be the case in most practical uses of our trace slicing algorithm, and when the corresponding parameter names are clear from context, we take the liberty to write partial functions compactly by simply listing their parameter values; for example, we write a partial function θ with $\theta(a) = a_2$, $\theta(b) = b_1$ and $\theta(c) = c_1$ as the sequence "$a_2b_1c_1$". The function $\perp$ then corresponds to the empty sequence.*

Example. Here is a parametric trace with events parametric in $\{a, b, c\}$ taking values in $\{a_1, a_2, b_1, c_1\}$: $\tau = e_1\langle a_1\rangle\ e_2\langle a_2\rangle\ e_3\langle b_1\rangle\ e_4\langle a_2b_1\rangle\ e_5\langle a_1\rangle\ e_6\langle\rangle\ e_7\langle b_1\rangle\ e_8\langle c_1\rangle\ e_9\langle a_2c_1\rangle\ e_{10}\langle a_1b_1c_1\rangle\ e_{11}\langle\rangle$. It may be the case that some of the base events appearing in a trace are the same; for example, e_1 may be equal to e_2 and to e_5. It is actually frequently the case in practice (at least in PQL [13], Tracematches [1], JavaMOP [9]) that parametric events are specified apriori with a given (sub)set of parameters, so that each event in $\mathcal{E}$ is always instantiated with partial functions over the same domain, that is, if $e\langle\theta\rangle$ and $e\langle\theta'\rangle$ appear in some parametric trace, then $\mathsf{Dom}(\theta) = \mathsf{Dom}(\theta')$. While this restriction is reasonable, our trace slicing and monitoring algorithms do not need it. □

Recall from Definition 4 that the trace slice $\tau\!\restriction_\theta$ keeps from τ only those events that are relevant for θ and drops their parameters.

Example. Consider again the sample parametric trace above with events parametric in $\{a, b, c\}$: $\tau = e_1\langle a_1\rangle\ e_2\langle a_2\rangle\ e_3\langle b_1\rangle\ e_4\langle a_2 b_1\rangle\ e_5\langle a_1\rangle\ e_6\langle\rangle\ e_7\langle b_1\rangle\ e_8\langle c_1\rangle\ e_9\langle a_2 c_1\rangle\ e_{10}\langle a_1 b_1 c_1\rangle\ e_{11}\langle\rangle$. Several slices of τ are listed below:

$$\begin{array}{lcl}
\tau\!\restriction_{a_1} & = & e_1 e_5 e_6 e_{11} \\
\tau\!\restriction_{a_2} & = & e_2 e_6 e_{11} \\
\tau\!\restriction_{a_1 b_1} & = & e_1 e_3 e_5 e_6 e_7 e_{11} \\
\tau\!\restriction_{a_2 b_1} & = & e_2 e_3 e_4 e_6 e_7 e_{11} \\
\tau\!\restriction & = & e_6 e_{11} \\
\tau\!\restriction_{a_1 b_1 c_1} & = & e_1 e_3 e_5 e_6 e_7 e_8 e_{10} e_{11} \\
\tau\!\restriction_{a_2 b_1 c_1} & = & e_2 e_3 e_4 e_6 e_7 e_8 e_9 e_{11} \\
\tau\!\restriction_{a_1 b_2 c_1} & = & e_1 e_5 e_6 e_8 e_{11} \\
\tau\!\restriction_{b_2 c_2} & = & e_6 e_{11}
\end{array}$$

In order for the partial functions above to make sense, we assumed that the set V_X in which parameters $X = \{a, b, c\}$ take values includes $\{a_1, a_2, b_1, b_2, c_1, c_2\}$. □

DEFINITION 13. *Given parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$, we let Θ_τ denote the lub closure of all the parameter instances appearing in events in τ, that is,* $\Theta_\tau = \overline{\{\theta \mid \theta \in [X \overset{\circ}{\rightarrow} V_X],\ e\langle\theta\rangle \in \tau\}}$.

PROPOSITION 12. *Θ_τ is a finite lub closed set for any $\tau \in \mathcal{E}\langle X\rangle^*$.*

Proof. Θ_τ is already defined as a lub closed set; since τ is finite, Corollary 3 implies that Θ_τ is finite. □

PROPOSITION 13. *Given $\tau\, e\langle\theta\rangle \in \mathcal{E}\langle X\rangle^*$, the following equality holds:* $\Theta_{\tau\, e\langle\theta\rangle} = \{\bot, \theta\} \sqcup \Theta_\tau$.

Proof. It follows by the following sequence of equalities:

$$\begin{array}{lcl}
\Theta_{\tau\, e\langle\theta\rangle} & = & \overline{\{\theta' \mid \theta' \in [X \overset{\circ}{\rightarrow} V_X],\ e'\langle\theta'\rangle \in \tau\, e\langle\theta\rangle\}} \\
& = & \overline{\{\theta\} \cup \{\theta' \mid \theta' \in [X \overset{\circ}{\rightarrow} V_X],\ e'\langle\theta'\rangle \in \tau\}} \\
& = & \overline{\{\theta\} \cup \Theta_\tau} \\
& = & \{\bot, \theta\} \sqcup \overline{\Theta_\tau} \\
& = & \{\bot, \theta\} \sqcup \Theta_\tau .
\end{array}$$

The first equality follows by Definition 13, the second by separating the case $e'\langle\theta'\rangle = e\langle\theta\rangle$, the third again by Definition 13, the fourth by Corollary 2, and the fifth by Proposition 12. Therefore, $\Theta_{\tau\, e\langle\theta\rangle}$ is the smallest lub closed set that contains θ and includes Θ_τ. □

PROPOSITION 14. *Given $\tau \in \mathcal{E}\langle X\rangle^*$ and $\theta \in [X \overset{\circ}{\rightarrow} V_X]$, the following equality holds:* $\tau\!\restriction_\theta = \tau\!\restriction_{\max(\theta]_{\Theta_\tau}}$.

Proof. We prove the following more general result:

> "let $\Theta \subseteq [X \overset{\circ}{\rightarrow} V_X]$ be lub closed and let $\theta \in [X \overset{\circ}{\rightarrow} V_X]$;
> then $\tau\!\restriction_\theta = \tau\!\restriction_{\max(\theta]_\Theta}$ for any $\tau \in \mathcal{E}\langle X\rangle^*$ with $\Theta_\tau \subseteq \Theta$."

First note that the statement above is well-formed because $\max(\theta]_\Theta$ exists whenever Θ is lub closed (*1.* in Proposition 8), and that it is indeed more general than the stated result: for the given $\tau \in \mathcal{E}\langle X\rangle^*$ and $\theta \in [X \overset{\circ}{\rightarrow} V_X]$, we pick Θ to be Θ_τ. We prove the general result by induction on the *length* of τ:

- If $|\tau| = 0$ then $\tau = \epsilon$ and $\epsilon\!\restriction_\theta = \epsilon\!\restriction_{\max(\theta]_\Theta} = \epsilon$.

- Now suppose that $\tau\!\restriction_\theta = \tau\!\restriction_{\max(\theta]_\Theta}$ for any $\tau \in \mathcal{E}\langle X\rangle^*$ with $\Theta_\tau \subseteq \Theta$ and $|\tau| = n \geq 0$, and let us show that $\tau'\!\restriction_\theta = \tau'\!\restriction_{\max(\theta]_\Theta}$ for any $\tau' \in \mathcal{E}\langle X\rangle^*$ with $\Theta_{\tau'} \subseteq \Theta$ and $|\tau'| = n+1$. Pick such a τ' and let $\tau' = \tau\, e\langle\theta'\rangle$ for a $\tau \in \mathcal{E}\langle X\rangle^*$ with $|\tau| = n$ and an $e\langle\theta'\rangle \in \mathcal{E}\langle X\rangle$. Since $\Theta_{\tau'} \subseteq \Theta$, by *6.* in Proposition 4 and by Proposition 13 it follows that $\Theta_\tau \subseteq \{\bot, \theta'\} \sqcup \Theta_\tau \subseteq \Theta$, so the induction hypothesis implies $\tau\!\restriction_\theta = \tau\!\restriction_{\max(\theta]_\Theta}$. The rest follows noticing that $\theta' \sqsubseteq \theta$ iff $\theta' \sqsubseteq \max(\theta]_\Theta$, which is a consequence of the definition of $\max(\theta]_\Theta$ because $\theta' \in \{\bot, \theta'\} \subseteq \{\bot, \theta'\} \sqcup \Theta_\tau \subseteq \Theta$ (again by *6.* in Proposition 4 and by Proposition 13).

Alternatively, one could have also done the proof above by induction on τ, not on its length, but the proof would be more involved, because one would need to prove that the domain over which the property is universally quantified, namely "any $\tau \in \mathcal{E}\langle X\rangle^*$ with $\Theta_\tau \subseteq \Theta$" is inductively generated. We therefore preferred to choose a more elementary induction schema. □

```
Algorithm 𝔸⟨X⟩
Input: parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$
Output: map $\mathbb{T} \in [[X \overset{\circ}{\rightarrow} V_X] \overset{\circ}{\rightarrow} \mathcal{E}^*]$ and set $\Theta \subseteq [X \overset{\circ}{\rightarrow} V_X]$

1 $\mathbb{T} \leftarrow \bot$; $\mathbb{T}(\bot) \leftarrow \epsilon$; $\Theta \leftarrow \{\bot\}$
2 foreach parametric event $e\langle\theta\rangle$ in order (fist to last) in $\tau$ do
3     foreach $\theta' \in \{\theta\} \sqcup \Theta$ do
4         $\mathbb{T}(\theta') \leftarrow \mathbb{T}(\max(\theta']_\Theta)\, e$
5     endfor
6     $\Theta \leftarrow \{\bot, \theta\} \sqcup \Theta$
7 endfor
```

Figure 1. Parametric trace slicing algorithm $\mathbb{A}\langle X\rangle$.

6. Algorithm for Parametric Trace Slicing

We next define an algorithm $\mathbb{A}\langle X\rangle$ that takes a parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$ incrementally (i.e., event by event), and builds a partial function $\mathbb{T} \in [[X \overset{\circ}{\rightarrow} V_X] \overset{\circ}{\rightarrow} \mathcal{E}^*]$ of finite domain that serves as a quick lookup table for all slices of τ. More precisely, Theorem 1 shows that, for any $\theta \in [X \overset{\circ}{\rightarrow} V_X]$, the trace slice $\tau\!\restriction_\theta$ is $\mathbb{T}(\max(\theta]_\Theta)$ after $\mathbb{A}\langle X\rangle$ processes τ, where $\Theta = \Theta_\tau$ is the domain of $\mathbb{T}$, a finite lub closed set of partial functions also calculated by $\mathbb{A}\langle X\rangle$ incrementally. Therefore, assuming that $\mathbb{A}\langle X\rangle$ is run on trace τ, all one has to do in order to calculate a slice $\tau\!\restriction_\theta$ for a given $\theta \in [X \overset{\circ}{\rightarrow} V_X]$ is to calculate $\max(\theta]_\Theta$ followed by a lookup into $\mathbb{T}$. This way the trace τ, 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 θ, without having to traverse the trace τ again, as an algorithm blindly following the definition of trace slicing would do.

Figure 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, making use of almost all the mathematical machinery developed so far in the paper. The algorithm $\mathbb{A}\langle X\rangle$ on input τ, written more succinctly $\mathbb{A}\langle X\rangle(\tau)$, traverses τ 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 Θ. After processing each event, the relationship between $\mathbb{T}$ and Θ 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, Θ 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, as in most online runtime verification systems. When a new event is received, say $e\langle\theta\rangle$, the mapping $\mathbb{T}$ is updated as follows: for each $\theta' \in [X \overset{\circ}{\rightarrow} V_X]$ that can be obtained by combining θ 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 $\mathbb{T}$

$e_1\langle a_1\rangle$	$e_2\langle a_2\rangle$	$e_3\langle b_1\rangle$	$e_4\langle a_2b_1\rangle$
$\langle\rangle{:}\,\epsilon$	$\langle\rangle{:}\,\epsilon$	$\langle\rangle{:}\,\epsilon$	$\langle\rangle{:}\,\epsilon$
$\langle a_1\rangle{:}\,e_1$	$\langle a_1\rangle{:}\,e_1$	$\langle a_1\rangle{:}\,e_1$	$\langle a_1\rangle{:}\,e_1$
	$\langle a_2\rangle{:}\,e_2$	$\langle a_2\rangle{:}\,e_2$	$\langle a_2\rangle{:}\,e_2$
		$\langle b_1\rangle{:}\,e_3$	$\langle b_1\rangle{:}\,e_3$
		$\langle a_1b_1\rangle{:}\,e_1e_3$	$\langle a_1b_1\rangle{:}\,e_1e_3$
		$\langle a_2b_1\rangle{:}\,e_2e_3$	$\langle a_2b_1\rangle{:}\,e_2e_3e_4$

$e_5\langle a_1\rangle$	$e_6\langle\rangle$	$e_7\langle b_1\rangle$
$\langle\rangle{:}\,\epsilon$	$\langle\rangle{:}\,e_6$	$\langle\rangle{:}\,e_6$
$\langle a_1\rangle{:}\,e_1e_5$	$\langle a_1\rangle{:}\,e_1e_5e_6$	$\langle a_1\rangle{:}\,e_1e_5e_6$
$\langle a_2\rangle{:}\,e_2$	$\langle a_2\rangle{:}\,e_2e_6$	$\langle a_2\rangle{:}\,e_2e_6$
$\langle b_1\rangle{:}\,e_3$	$\langle b_1\rangle{:}\,e_3e_6$	$\langle b_1\rangle{:}\,e_3e_6e_7$
$\langle a_1b_1\rangle{:}\,e_1e_3e_5$	$\langle a_1b_1\rangle{:}\,e_1e_3e_5e_6$	$\langle a_1b_1\rangle{:}\,e_1e_3e_5e_6e_7$
$\langle a_2b_1\rangle{:}\,e_2e_3e_4$	$\langle a_2b_1\rangle{:}\,e_2e_3e_4e_6$	$\langle a_2b_1\rangle{:}\,e_2e_3e_4e_6e_7$

$e_8\langle c_1\rangle$	$e_9\langle a_2c_1\rangle$	$e_{10}\langle a_1b_1c_1\rangle$	$e_{11}\langle\rangle$
$\langle\rangle{:}\,e_6$	$\langle\rangle{:}\,e_6$	$\langle\rangle{:}\,e_6$	$\langle\rangle{:}\,e_6e_{11}$
$\langle a_1\rangle{:}\,e_1e_5e_6$	$\langle a_1\rangle{:}\,e_1e_5e_6$	$\langle a_1\rangle{:}\,e_1e_5e_6$	$\langle a_1\rangle{:}\,e_1e_5e_6e_{11}$
$\langle a_2\rangle{:}\,e_2e_6$	$\langle a_2\rangle{:}\,e_2e_6$	$\langle a_2\rangle{:}\,e_2e_6$	$\langle a_2\rangle{:}\,e_2e_6e_{11}$
$\langle b_1\rangle{:}\,e_3e_6e_7$	$\langle b_1\rangle{:}\,e_3e_6e_7$	$\langle b_1\rangle{:}\,e_3e_6e_7$	$\langle b_1\rangle{:}\,e_3e_6e_7e_{11}$
$\langle a_1b_1\rangle{:}\,e_1e_3e_5e_6e_7$	$\langle a_1b_1\rangle{:}\,e_1e_3e_5e_6e_7$	$\langle a_1b_1\rangle{:}\,e_1e_3e_5e_6e_7$	$\langle a_1b_1\rangle{:}\,e_1e_3e_5e_6e_7e_{11}$
$\langle a_2b_1\rangle{:}\,e_2e_3e_4e_6e_7$	$\langle a_2b_1\rangle{:}\,e_2e_3e_4e_6e_7$	$\langle a_2b_1\rangle{:}\,e_2e_3e_4e_6e_7$	$\langle a_2b_1\rangle{:}\,e_2e_3e_4e_6e_7e_{11}$
$\langle c_1\rangle{:}\,e_6e_8$	$\langle c_1\rangle{:}\,e_6e_8$	$\langle c_1\rangle{:}\,e_6e_8$	$\langle c_1\rangle{:}\,e_6e_8e_{11}$
$\langle a_1c_1\rangle{:}\,e_1e_5e_6e_8$	$\langle a_1c_1\rangle{:}\,e_1e_5e_6e_8$	$\langle a_1c_1\rangle{:}\,e_1e_5e_6e_8$	$\langle a_1c_1\rangle{:}\,e_1e_5e_6e_8e_{11}$
$\langle a_2c_1\rangle{:}\,e_2e_6e_8$	$\langle a_2c_1\rangle{:}\,e_2e_6e_8e_9$	$\langle a_2c_1\rangle{:}\,e_2e_6e_8e_9$	$\langle a_2c_1\rangle{:}\,e_2e_6e_8e_9e_{11}$
$\langle b_1c_1\rangle{:}\,e_3e_6e_7e_8$	$\langle b_1c_1\rangle{:}\,e_3e_6e_7e_8$	$\langle b_1c_1\rangle{:}\,e_3e_6e_7e_8$	$\langle b_1c_1\rangle{:}\,e_3e_6e_7e_8e_{11}$
$\langle a_1b_1c_1\rangle{:}\,e_1e_3e_5e_6e_7e_8$	$\langle a_1b_1c_1\rangle{:}\,e_1e_3e_5e_6e_7e_8$	$\langle a_1b_1c_1\rangle{:}\,e_1e_3e_5e_6e_7e_8e_{10}$	$\langle a_1b_1c_1\rangle{:}\,e_1e_3e_5e_6e_7e_8e_{10}e_{11}$
$\langle a_2b_1c_1\rangle{:}\,e_2e_3e_4e_6e_7e_8$	$\langle a_2b_1c_1\rangle{:}\,e_2e_3e_4e_6e_7e_8e_9$	$\langle a_2b_1c_1\rangle{:}\,e_2e_3e_4e_6e_7e_8e_9$	$\langle a_2b_1c_1\rangle{:}\,e_2e_3e_4e_6e_7e_8e_9e_{11}$

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

that is less informative or as informative as θ'; the Θ data-structure is then extended in line 6.

Example. Consider again the sample parametric trace above with events parametric in $\{a, b, c\}$, namely $\tau = e_1\langle a_1\rangle\ e_2\langle a_2\rangle\ e_3\langle b_1\rangle\ e_4\langle a_2b_1\rangle\ e_5\langle a_1\rangle\ e_6\langle\rangle\ e_7\langle b_1\rangle\ e_8\langle c_1\rangle\ e_9\langle a_2c_1\rangle\ e_{10}\langle a_1b_1c_1\rangle\ e_{11}\langle\rangle$. Table 6 shows how $\mathbb{A}\langle X\rangle$ works on τ. 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 undefined on any other partial function. Obviously, the Θ corresponding to a cell is the union of all the θ's that appear in pairs "$\langle\theta\rangle : w$" in that cell. 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, and that the Θ corresponding to each cell is lub closed. □

$\mathbb{A}\langle X\rangle$ 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 τ) proof of correctness for $\mathbb{A}\langle X\rangle$, but we prefer to keep this discussion informal and give a rigorous proof shortly after. The role of this discussion is twofold: (1) to better explain the algorithm $\mathbb{A}\langle X\rangle$, providing the reader with additional intuition for its difficulty and compactness, and (2) to give a proof sketch for the correctness of $\mathbb{A}\langle X\rangle$.

Let us first note that a partial function added to Θ will never be removed from Θ; 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 θ is processed by $\mathbb{A}\langle X\rangle$. Moreover, since $\mathsf{Dom}(\mathbb{T}) = \Theta = \Theta_\epsilon = \{\bot\}$ initially and since *5.* and *7.* in Proposition 4 imply $\Theta \cup (\{\theta\} \sqcup \Theta) = \{\bot, \theta\} \sqcup \Theta$ while Proposition 13 states that $\Theta_{\tau\, e\langle\theta\rangle} = \{\bot, \theta\} \sqcup \Theta_\tau$, 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 τ.

Each θ' considered by the loop at lines 3-5 has the property that $\theta \sqsubseteq \theta'$, and at (precisely) one iteration of the loop θ' is θ; indeed, $\theta \in \{\theta\}\sqcup\Theta$ because $\bot \in \Theta$. Thanks to Proposition 14, the claimed Theorem 1 holds essentially 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 *3.* in Proposition 8 implies 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 $\tau{\restriction}_{\theta'}$ by Proposition 14. 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 τ. This concludes our informal proof sketch; let us next give a rigorous proof of correctness for our trace slicing algorithm.

DEFINITION 14. *Let $\mathbb{A}\langle X\rangle(\tau).\mathbb{T}$ and $\mathbb{A}\langle X\rangle(\tau).\Theta$ be the two data-structures ($\mathbb{T}$ and Θ) of $\mathbb{A}\langle X\rangle$ after it processes τ.*

THEOREM 1. *The following hold for any $\tau \in \mathcal{E}\langle X\rangle^*$:*
1. *$\mathsf{Dom}(\mathbb{A}\langle X\rangle(\tau).\mathbb{T}) = \mathbb{A}\langle X\rangle(\tau).\Theta = \Theta_\tau$;*
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 $\theta \in [X \overset{\circ}{\to} V_X]$.*

Proof. Since $\mathbb{A}\langle X\rangle$ processes the events in the input trace in order, when given the input $\tau\, e\langle\theta\rangle$, the Θ and $\mathbb{T}$ structures after $\mathbb{A}\langle X\rangle$ processes τ but before it processes $e\langle\theta\rangle$ (i.e., right before the last iteration of the loop at lines 2-7) are precisely $\mathbb{A}\langle X\rangle(\tau).\Theta$ and $\mathbb{A}\langle X\rangle(\tau).\mathbb{T}$, respectively. Further, the loop at lines 3-5 updates $\mathbb{T}$ on all $\theta' \in \{\theta\} \sqcup \Theta$; in case $\mathbb{T}$ was not defined on such a θ', then it will be defined after $e\langle\theta\rangle$ is processed. The definitional domain of $\mathbb{T}$ is thus continuously growing or potentially remains stationary as parametric events are processed, but it never decreases.

With these observations, we can prove *1.* easily by induction on τ. If $\tau = \epsilon$ then $\mathsf{Dom}(\mathbb{A}\langle X\rangle(\epsilon).\mathbb{T}) = \mathbb{A}\langle X\rangle(\epsilon).\Theta = \Theta_\epsilon = \{\bot\}$. Suppose now that $\mathsf{Dom}(\mathbb{A}\langle X\rangle(\tau).\mathbb{T}) = \mathbb{A}\langle X\rangle(\tau).\Theta = \Theta_\tau$ holds for $\tau \in \mathcal{E}\langle X\rangle^*$, and let $e\langle\theta\rangle \in \mathcal{E}\langle X\rangle$ be any parametric event. Then the following concludes the proof of *1.*:

$$\begin{array}{rl}
& \mathsf{Dom}(\mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\mathbb{T}) \\
= & \mathsf{Dom}(\mathbb{A}\langle X\rangle(\tau).\mathbb{T}) \cup (\{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta) \\
= & \mathbb{A}\langle X\rangle(\tau).\Theta \cup (\{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta) \\
= & (\{\bot\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta) \cup (\{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta) \\
= & \{\bot, \theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta \\
= & \mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\Theta \\
= & \{\bot, \theta\} \sqcup \Theta_\tau \\
= & \Theta_{\tau\, e\langle\theta\rangle}
\end{array}$$

where the first equality follows from how the loop at lines 3-5 updates $\mathbb{T}$, the second by the induction hypothesis, the third by *5.* in Proposition 4, the fourth by *7.* in Proposition 4, the fifth by how Θ is updated at line 6, the sixth again by the induction hypothesis, and, finally, the seventh by Proposition 13.

Before we continue, let us first prove the following property:

$$\mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\mathbb{T}(\theta') = \mathbb{A}\langle X\rangle(\tau).\mathbb{T}(\max\,(\theta']_{\mathbb{A}\langle X\rangle(\tau).\Theta})\, e$$
for any $e\langle\theta\rangle \in \mathcal{E}\langle X\rangle$ and any $\theta' \in \{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta$.

One should be careful here to *not* get tricked thinking that this property is straightforward, because it says only what line 4 of $\mathbb{A}\langle X\rangle$ does. The complexity comes from the fact that if there were two different $\theta_1, \theta_2 \in \{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta$ such that $\theta_1 = \max\,(\theta_2]_{\mathbb{A}\langle X\rangle(\tau).\Theta}$, then an unfortunate enumeration of the partial functions θ' in $\{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta$ by the loop at lines 3-5 may lead to the non-parametric event e to be added twice to a slice: indeed, if θ_1 is processed before θ_2, then e is first added to the end of $\mathbb{T}(\theta_1)$ when $\theta' = \theta_1$, and then $\mathbb{T}(\theta_1)\, e$ is assigned to $\mathbb{T}(\theta_2)$ when $\theta' = \theta_2$; this way, $\mathbb{T}(\theta_2)$ ends up accumulating e twice instead of once, which is obviously wrong. Fortunately, since $\mathbb{A}\langle X\rangle(\tau).\Theta$ is lub closed (by *1.* above and Proposition 12), *3.* in Proposition 8 implies that there are no such different $\theta_1, \theta_2 \in \{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta$. Therefore, there is no interference between the various assignments at line 4, regardless of the order in which the partial functions $\theta' \in \{\theta\} \sqcup \Theta$ are enumerated, which means that, indeed, $\mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\mathbb{T}(\theta') = \mathbb{A}\langle X\rangle(\tau).\mathbb{T}(\max{(\theta']_{\mathbb{A}\langle X\rangle(\tau).\Theta}})\, e$ for any $e\langle\theta\rangle \in \mathcal{E}\langle X\rangle$ and for any $\theta' \in \{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta$. This lack of interference between updates of $\mathbb{T}$ also suggests an important implementation optimization: *the loop at lines 3-5 can be parallelized without having to duplicate the table* $\mathbb{T}$*!* Of course, the loop can be parallelized anyway if the table is duplicated and then merged within the original table, in the sense that all the writes to $\mathbb{T}(\theta')$ are done in a copy of $\mathbb{T}$. However, experiments show that the table $\mathbb{T}$ can be literally huge in real applications, in the order of billions of entries, so duplicating and merging it can be prohibitive.

2. can be now proved by induction on the length of τ. If $\tau = \epsilon$ then $\mathbb{A}\langle X\rangle(\epsilon).\Theta = \{\bot\}$, so $\theta' \in \mathbb{A}\langle X\rangle(\epsilon).\Theta$ can only be $\bot$; then $\mathbb{A}\langle X\rangle(\epsilon).\mathbb{T}(\bot) = \tau\upharpoonright_{\bot} = \epsilon$. Suppose now that $\mathbb{A}\langle X\rangle(\tau).\mathbb{T}(\theta') = \tau\upharpoonright_{\theta'}$ for any $\theta' \in \mathbb{A}\langle X\rangle(\tau).\Theta$ and let us show that $\mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\mathbb{T}(\theta') = (\tau\, e\langle\theta\rangle)\upharpoonright_{\theta'}$ for any $\theta' \in \mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\Theta$. As shown in the proof of *1.* above, $\mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\Theta = \mathbb{A}\langle X\rangle(\tau).\Theta \cup (\{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta)$, so we have two cases to analyze. First, if $\theta' \in \{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta$ then $\theta \sqsubseteq \theta'$ and so $(\tau\, e\langle\theta\rangle)\upharpoonright_{\theta'} = \tau\upharpoonright_{\theta'}\, e$; further,

$$\begin{array}{rcl}\mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\mathbb{T}(\theta') & = & \mathbb{A}\langle X\rangle(\tau).\mathbb{T}(\max{(\theta']_{\mathbb{A}\langle X\rangle(\tau).\Theta}})\, e \\ & = & \tau\upharpoonright_{\max{(\theta']_{\mathbb{A}\langle X\rangle(\tau).\Theta}}}\ e \\ & = & \tau\upharpoonright_{\theta'}\ e \\ & = & (\tau\, e\langle\theta\rangle)\upharpoonright_{\theta'},\end{array}$$

where the first equality follows by the auxiliary property proved above, the second by the induction hypothesis using the fact that $\max{(\theta']_{\mathbb{A}\langle X\rangle(\tau).\Theta}} \in \mathbb{A}\langle X\rangle(\tau).\Theta$, and the third by Proposition 14.

Second, if $\theta' \in \mathbb{A}\langle X\rangle(\tau).\Theta$ but $\theta' \notin \{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta$ then $\theta \not\sqsubseteq \theta'$ and so $(\tau\, e\langle\theta\rangle)\upharpoonright_{\theta'} = \tau\upharpoonright_{\theta'}$; further,

$$\begin{array}{rcl}\mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\mathbb{T}(\theta') & = & \mathbb{A}\langle X\rangle(\tau).\mathbb{T}(\theta') \\ & = & \tau\upharpoonright_{\theta'} \\ & = & (\tau\, e\langle\theta\rangle)\upharpoonright_{\theta'},\end{array}$$

where the first equality holds because θ' is not considered by the loop in lines 3-5 in $\mathbb{A}\langle X\rangle$, that is, $\theta' \notin \{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta$, and the second equality follows by the induction hypothesis, as $\theta' \in \mathbb{A}\langle X\rangle(\tau).\Theta$. Therefore, $\mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\mathbb{T}(\theta') = (\tau\, e\langle\theta\rangle)\upharpoonright_{\theta'}$ for any $\theta' \in \mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\Theta$, which completes the proof of *2.*

3. is the main result concerning our trace slicing algorithm and it follows now easily:

$$\begin{array}{rcl}\tau\upharpoonright_{\theta} & = & \tau\upharpoonright_{\max{(\theta]_{\Theta_\tau}}} \\ & = & \tau\upharpoonright_{\max{(\theta]_{\mathbb{A}\langle X\rangle(\tau).\Theta}}} \\ & = & \mathbb{A}\langle X\rangle(\tau).\mathbb{T}(\max{(\theta]_{\mathbb{A}\langle X\rangle(\tau).\Theta}})\end{array}$$

The first equality follows by Proposition 14, the second by *1.* and the third by *2.*, as $\max{(\theta]_{\mathbb{A}\langle X\rangle(\tau).\Theta}} \in \mathbb{A}\langle X\rangle(\tau).\Theta$. This concludes the correctness proof of our trace slicing algorithm $\mathbb{A}\langle X\rangle$. $\square$

7. Parametric Monitors

In this section we first define monitors M as a variant of Moore machines with potentially infinitely many states; then we define parametric monitors $\Lambda X.M$ as monitors maintaining one state of M per parameter instance. Like for parametric properties, which turned out to be 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 also 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.

7.1 The Non-Parametric Case

We start by defining non-parametric monitors as a variant of Moore machines [15] that allows infinitely many states:

DEFINITION 15. *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, σ is the transition function, and γ is the output function. The transition function is extended to $\sigma : S \times \mathcal{E}^* \to S$ as expected: $\sigma(s, \epsilon) = s$ and $\sigma(s, we) = \sigma(\sigma(s, w), e)$ for any $s \in S$, $e \in \mathcal{E}$, and $w \in \mathcal{E}^*$.*

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. Consider, for example, a monitor for a property specified using an NFA: the monitor performs an NFA-to-DFA construction on the fly, as events are received, thus generating only those states in the DFA that are needed by the monitored execution trace; since generation of next set of states is fast, one need not even hash the generated DFA states, the entire memory needed by monitor staying linear in the size of the NFA.

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 states are reached. For example, monitors for context-free grammars like the ones in [14] 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. What is common to all monitors though is that they can take a trace event-by-event and, as each event is processed, classify the observed trace into a category. The following is natural:

DEFINITION 16. *$M = (S, \mathcal{E}, \mathcal{C}, \imath, \sigma, \gamma)$ is a* ***monitor for property*** *$P : \mathcal{E}^* \to \mathcal{C}$ iff $\gamma(\sigma(\imath, w)) = P(w)$ for each $w \in \mathcal{E}^*$.*

Since we allow monitors to have infinitely many states, there is a strong correspondence between properties and monitors:

PROPOSITION 15. *Every monitor M defines a property $\mathcal{P}_M$ with M a monitor for $\mathcal{P}_M$. Every property P defines a monitor $\mathcal{M}_P$ with $\mathcal{M}_P$ a monitor for P. For any property P, $\mathcal{P}_{\mathcal{M}_P} = P$.*

Proof. Given $M = (S, \mathcal{E}, \mathcal{C}, \imath, \sigma, \gamma)$, let $\mathcal{P}_M : \mathcal{E}^* \to \mathcal{C}$ be the property $\mathcal{P}_M(w) = \gamma(\sigma(\imath, w))$; note that M is indeed a monitor for $\mathcal{P}_M$. Given $P : \mathcal{E}^* \to \mathcal{C}$, let $\mathcal{M}_P$ be the monitor $(S_P, \mathcal{E}, \mathcal{C}, \imath_P, \sigma_P, \gamma_P)$ with $S_P = \mathcal{E}^*$, $\imath_P = \epsilon$, $\sigma_P(w, e) = we$, $\gamma_P(w) = P(w)$. $\mathcal{M}_P$ is a monitor for P as $\gamma_P(\sigma_P(\imath_P, w)) = \gamma_P(\sigma_P(\epsilon, w)) = \gamma_P(\epsilon w) = \gamma_P(w) = P(w)$. Finally, $\mathcal{P}_{\mathcal{M}_P}(w) = \gamma_P(\sigma_P(\imath_P, w)) = P(w)$ for any $w \in \mathcal{E}^*$, so $\mathcal{P}_{\mathcal{M}_P} = P$. $\square$

The equality of monitors $\mathcal{M}_{\mathcal{P}_M} = M$ does not hold for any monitor M; it does hold when $M = \mathcal{M}_P$ for some property P.

DEFINITION 17. *Monitors M and M' are* ***property equivalent****, or just* ***equivalent****, written $M \equiv M'$, iff they are monitors for the same property; with the notation above, $M \equiv M'$ iff $\mathcal{P}_M = \mathcal{P}_{M'}$.*

COROLLARY 4. *With the notation in Proposition 15, $\mathcal{M}_{\mathcal{P}_M} \equiv M$.*

7.2 The Parametric Case

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 18. *Given parameters X with corresponding values V_X and monitor $M = (S, \mathcal{E}, \mathcal{C}, \imath, \sigma : S \times \mathcal{E} \to S, \gamma : S \to \mathcal{C})$, we define the* ***parametric monitor*** *$\Lambda X.M$ as the monitor*

$$([[X \overset{\circ}{\to} V_X] \to S], \mathcal{E}\langle X\rangle, [[X \overset{\circ}{\to} V_X] \to \mathcal{C}], \lambda\theta.\imath, \Lambda X.\sigma, \Lambda X.\gamma),$$

with $\Lambda X.\sigma : [[X \overset{\circ}{\to} V_X] \to S] \times \mathcal{E}\langle X\rangle \to [[X \overset{\circ}{\to} V_X] \to S]$ and $\Lambda X.\gamma : [[X \overset{\circ}{\to} V_X] \to S] \to [[X \overset{\circ}{\to} V_X] \to \mathcal{C}]$ 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 $\delta \in [[X \overset{\circ}{\to} V_X] \to S]$ and any $\theta, \theta' \in [X \overset{\circ}{\to} V_X]$.

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

PROPOSITION 16. *If M is a monitor for property P then parametric monitor $\Lambda X.M$ is a monitor for parametric property $\Lambda X.P$, or, with the notation in Proposition 15, $\mathcal{P}_{\Lambda X.M} = \Lambda X.\mathcal{P}_M$.*

Proof. We show that $(\Lambda X.\gamma)((\Lambda X.\sigma)(\lambda\theta.\imath, \tau)) = (\Lambda X.P)(\tau)$ for any $\tau \in \mathcal{E}\langle X\rangle^*$, i.e., after application on $\theta \in [X \overset{\circ}{\to} V_X]$, that $\gamma((\Lambda X.\sigma)(\lambda\theta.\imath, \tau)(\theta)) = P(\tau\upharpoonright_\theta)$ for any $\tau \in \mathcal{E}\langle X\rangle^*$ and $\theta[X \overset{\circ}{\to} V_X]$. Since M is a monitor for P, it suffices to show that $(\Lambda X.\sigma)(\lambda\theta.\imath, \tau)(\theta) = \sigma(\imath, \tau\upharpoonright_\theta)$ for any $\tau \in \mathcal{E}\langle X\rangle^*$ and $\theta[X \overset{\circ}{\to} V_X]$. We prove it by induction on τ. If $\tau = \epsilon$ then $(\Lambda X.\sigma)(\lambda\theta.\imath, \epsilon)(\theta) = (\lambda\theta.\imath)(\theta) = \imath = \sigma(\imath, \epsilon) = \sigma(\imath, \epsilon\upharpoonright_\theta)$. Suppose now that $(\Lambda X.\sigma)(\lambda\theta.\imath, \tau)(\theta) = \sigma(\imath, \tau\upharpoonright_\theta)$ for some arbitrary but fixed $\tau \in \mathcal{E}\langle X\rangle^*$ and for any $\theta \in [X \overset{\circ}{\to} V_X]$, and let $e\langle\theta'\rangle$ be any parametric event in $\mathcal{E}\langle X\rangle$ and let $\theta \in [X \overset{\circ}{\to} V_X]$ be any parameter instance. The inductive step is then as follows:

$$\begin{aligned}(\Lambda X.\sigma)(\lambda\theta.\imath, \tau\, e\langle\theta'\rangle)(\theta) &= (\Lambda X.\sigma)((\Lambda X.\sigma)(\lambda\theta.\imath, \tau), e\langle\theta'\rangle)(\theta) \\ &= (\Lambda X.\sigma)(\sigma(\imath, \tau\upharpoonright_\theta), e\langle\theta'\rangle)(\theta) \\ &= \begin{cases} \sigma(\sigma(\imath, \tau\upharpoonright_\theta), e) & \text{if } \theta' \sqsubseteq \theta \\ \sigma(\imath, \tau\upharpoonright_\theta) & \text{if } \theta' \not\sqsubseteq \theta \end{cases} \\ &= \begin{cases} \sigma(\imath, \tau\upharpoonright_\theta\ e) & \text{if } \theta' \sqsubseteq \theta \\ \sigma(\imath, \tau\upharpoonright_\theta) & \text{if } \theta' \not\sqsubseteq \theta \end{cases} \\ &= \sigma(\imath, (\tau\, e\langle\theta'\rangle)\upharpoonright_\theta)\end{aligned}$$

The first equality above follows by the second part of Definition 15), the second by the induction hypothesis, the third by Definition 18, the fourth again by the second part of Definition 15, and the fifth by Definition 4. This concludes our proof. □

8. Algorithm for Parametric Trace Monitoring

We next propose a monitoring algorithm for parametric properties. Analyzing the definition of a parametric monitor (Definition 18), the first thing we note is that its state space is not only infinite, but it is not even enumerable. Therefore, a first challenge in monitoring parametric properties is how to represent the states of the parametric monitor. Inspired by the algorithm for trace slicing in Figure 1, we encode functions $[[X \overset{\circ}{\to} V_X] \overset{\circ}{\to} S]$ as tables with entries indexed by parameter instances in $[X \overset{\circ}{\to} V_X]$ and with contents states in S. Following similar arguments as in the proof of the trace slicing algorithm, such tables will have a finite number of entries provided that each event instantiates only a finite number of parameters.

Figure 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 9 shows one way to use this algorithm: a monitor M is first synthesized from the base property P,

```
Algorithm $\mathbb{B}\langle X\rangle(M = (S, \mathcal{E}, \mathcal{C}, \imath, \sigma, \gamma))$
Input: finite parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$
Output: mapping $\Gamma : [[X \overset{\circ}{\to} V_X] \overset{\circ}{\to} \mathcal{C}]$ and set $\Theta \subseteq [X \overset{\circ}{\to} V_X]$

1 $\Delta \leftarrow \bot$; $\Delta(\bot) \leftarrow \imath$; $\Theta \leftarrow \{\bot\}$
2 foreach parametric event $e\langle\theta\rangle$ in order in $\tau$ do
3   foreach $\theta' \in \{\theta\} \sqcup \Theta$ do
4     $\Delta(\theta') \leftarrow \sigma(\Delta(\max(\theta']_\Theta), e)$
5     $\Gamma(\theta') \leftarrow \gamma(\Delta(\theta'))$       // a message may be output here
6   endfor
7   $\Theta \leftarrow \{\bot, \theta\} \sqcup \Theta$
8 endfor
```

Figure 2. Parametric monitoring algorithm $\mathbb{B}\langle X\rangle$

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 Figure 1, the main difference being that trace slices are processed, as generated, by M: instead of calculating the trace slice of θ' 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 Δ the state of the "monitor instance" corresponding to θ' 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 Γ. In other words, we replace trace slices in $\mathbb{A}\langle X\rangle$ by local monitors processing online those slices. In our implementation in Section 9, we also check whether $\Gamma(\theta')$ at line 5 violates the property and, if so, an error message including θ' is output to the user.

DEFINITION 19. *Given $\tau \in \mathcal{E}\langle X\rangle^*$, let $\mathbb{B}\langle X\rangle(M)(\tau).\Theta$ and $\mathbb{B}\langle X\rangle(M)(\tau).\Delta$ and $\mathbb{B}\langle X\rangle(M)(\theta).\Gamma$ be the three data-structures maintained by the algorithm $\mathbb{B}\langle X\rangle(M)$ in Figure 2 after processing τ. Let $\bot \mapsto \imath = \mathbb{B}\langle X\rangle(M)(\epsilon).\Delta \in [[X \overset{\circ}{\to} V_X] \overset{\circ}{\to} S]$ be the partial map taking $\bot \in [X \overset{\circ}{\to} V_X]$ to $\imath$ and undefined elsewhere.*

COROLLARY 5. *The following hold for any $\tau \in \mathcal{E}\langle X\rangle^*$:*

1. *Dom$(\mathbb{B}\langle X\rangle(M)(\tau).\Delta) = \mathbb{B}\langle X\rangle(M)(\tau).\Theta = \Theta_\tau$;*
2. *$\mathbb{B}\langle X\rangle(M)(\tau).\Delta(\theta) = \sigma(\imath, \tau\upharpoonright_\theta)$ and*
   $\mathbb{B}\langle X\rangle(M)(\tau).\Gamma(\theta) = \gamma(\sigma(\imath, \tau\upharpoonright_\theta))$
   for any $\theta \in \mathbb{B}\langle X\rangle(M)(\tau).\Theta$;
3. *$\sigma(\imath, \tau\upharpoonright_\theta) = \mathbb{B}\langle X\rangle(M)(\tau).\Delta(\max(\theta]_{\mathbb{B}\langle X\rangle(M)(\tau).\Theta})$ and*
   $\gamma(\sigma(\imath, \tau\upharpoonright_\theta)) = \mathbb{B}\langle X\rangle(M)(\tau).\Gamma(\max(\theta]_{\mathbb{B}\langle X\rangle(M)(\tau).\Theta})$
   for any $\theta \in [X \overset{\circ}{\to} V_X]$.

Proof. Follows from Theorem 1 and the discussion above. □

We next associate a monitor to the algorithm in Figure 2:

DEFINITION 20. *For the algorithm $\mathbb{B}\langle X\rangle(M)$ in Figure 2, let $\mathcal{M}_{\mathbb{B}\langle X\rangle(M)} = (R, \mathcal{E}\langle X\rangle, [[X \overset{\circ}{\to} V_X] \to \mathcal{C}], , \bot \mapsto \imath, next, out)$ be the monitor defined as follows: $R \subseteq [[X \overset{\circ}{\to} V_X] \overset{\circ}{\to} S]$ is the set $\{\mathbb{B}\langle X\rangle(M)(\tau).\Delta \mid \tau \in \mathcal{E}\langle X\rangle^*\}$ of reachable Δ's in $\mathbb{B}\langle X\rangle(M)$, and next $: R \times \mathcal{E}\langle X\rangle \to R$ and out $: R \to [[X \overset{\circ}{\to} V_X] \to \mathcal{C}]$ are the functions defined as follows ($\tau \in \mathcal{E}\langle X\rangle^*$, $e \in \mathcal{E}$, $\theta \in [X \overset{\circ}{\to} V_X]$):*

next$(\mathbb{B}\langle X\rangle(M)(\tau).\Delta, e\langle\theta\rangle) = \mathbb{B}\langle X\rangle(M)(\tau\, e\langle\theta\rangle).\Delta$, and
out$(\mathbb{B}\langle X\rangle(M)(\tau).\Delta)(\theta) = \mathbb{B}\langle X\rangle(M)(\tau).\Gamma(\max(\theta]_{\mathbb{B}\langle X\rangle(M)(\tau).\Theta})$.

THEOREM 2. *$\mathcal{M}_{\mathbb{B}\langle X\rangle(M)} \equiv \Lambda X.M$ for any monitor M.*

Proof. All we have to do is to show that, for any $\tau \in \mathcal{E}\langle X\rangle^*$, *out*(*next*($\bot \mapsto \imath, \tau$)) and $(\Lambda X.\gamma)((\Lambda X.\sigma)(\lambda\theta.\imath, \tau))$ are equal as

```
Algorithm $\mathbb{C}\langle X\rangle(M = (S, \mathcal{E}, \mathcal{C}, \imath, \sigma, \gamma))$
Globals: mapping $\Delta : [[X \overset{\circ}{\to} V_X] \overset{\circ}{\to} S]$ and
         mapping $\mathcal{U} : [X \overset{\circ}{\to} V_X] \to \mathcal{P}_f([X \overset{\circ}{\to} V_X])$ and
         mapping $\Gamma : [[X \overset{\circ}{\to} V_X] \overset{\circ}{\to} \mathcal{C}]$
Initialization: $\mathcal{U}(\theta) \leftarrow \emptyset$ for any $\theta \in [X \overset{\circ}{\to} V_X]$, $\Delta(\bot) \leftarrow \imath$

function main($e\langle\theta\rangle$)
 1 if $\Delta(\theta)$undefined then
 2     foreach $\theta_{max} \sqsubset \theta$ (in reversed topological order) do
 3         if $\Delta(\theta_{max})$ defined then
 4             goto 7
 5         endif
 6     endfor
 7     defineTo($\theta, \theta_{max}$)
 8     foreach $\theta_{max} \sqsubset \theta$ (in reversed topological order) do
 9         foreach $\theta_{comp} \in \mathcal{U}(\theta_{max})$ that is compatible with $\theta$ do
10             if $\Delta(\theta_{comp} \sqcup \theta)$ undefined then
11                 defineTo($\theta_{comp} \sqcup \theta, \theta_{comp}$)
12             endif
13         endfor
14     endfor
15 endif
16 foreach $\theta' \in \{\theta\} \cup \mathcal{U}(\theta)$ do
17     $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e)$
18     $\Gamma(\theta') \leftarrow \sigma(\Delta(\theta'))$
19 endfor

function defineTo($\theta, \theta'$)
1 $\Delta(\theta) \leftarrow \Delta(\theta')$
2 foreach $\theta'' \sqsubset \theta$ do
3     $\mathcal{U}(\theta'') \leftarrow \mathcal{U}(\theta'') \cup \{\theta\}$
4 endfor
```

Figure 3. Online parametric monitoring algorithm $\mathbb{C}\langle X\rangle$

total functions in $[[X \overset{\circ}{\to} V_X] \to \mathcal{C}]$. Let $\theta \in [X \overset{\circ}{\to} V_X]$; then:

$$\begin{aligned} \textit{out}(\textit{next}(\bot \mapsto \imath, \tau))(\theta) &= \textit{out}(\mathbb{B}\langle X\rangle(M)(\tau).\Delta)(\theta) \\ &= \mathbb{B}\langle X\rangle(M)(\tau).\Gamma(\max{(\theta]_{\mathbb{B}\langle X\rangle(M)(\tau).\Theta}}) \\ &= \gamma(\sigma(\lambda\theta.\imath, \tau\!\upharpoonright_\theta)) \\ &= \gamma((\Lambda X.\sigma)(\lambda\theta.\imath, \tau)(\theta)) \\ &= (\Lambda X.\gamma)((\Lambda X.\sigma)(\lambda\theta.\imath, \tau))(\theta). \end{aligned}$$

The first equality above follows inductively by the definition of *next* (Definition 20), noticing that $\bot \mapsto \imath = \mathbb{B}\langle X\rangle(M)(\epsilon).\Delta$. The second equality follows by the definition of *out* (Definition 20) and the third by *3.* in Corollary 5. The fourth equality above follows inductively by the definition of $\Lambda X.\sigma$ (Definition 18) and has already been proved as part of the proof of Proposition 16. Finally, the fifth equality follows by the definition of $\Lambda X.\gamma$ (Definition 18).

Therefore, $\mathcal{M}_{\mathbb{B}\langle X\rangle(M)}$ and $\Lambda X.M$ define the same property. □

COROLLARY 6. *If M is a monitor for P and X is a set of parameters, then $\mathcal{M}_{\mathbb{B}\langle X\rangle(M)}$ is a monitor for parametric property $\Lambda X.P$.*

Proof. With the notation in Proposition 15, Theorem 2 implies that $\mathcal{P}_{\mathcal{M}_{\mathbb{B}\langle X\rangle(M)}} = \mathcal{P}_{\Lambda X.M}$. By Proposition 16 and the fact that $P = \mathcal{P}_M$, we conclude that $\mathcal{P}_{\mathcal{M}_{\mathbb{B}\langle X\rangle(M)}} = \Lambda X.P$. □

9. Implementation and Evaluation

Algorithm $\mathbb{C}\langle X\rangle$ in Figure 3 refines Algorithm $\mathbb{B}\langle X\rangle$ in Figure 2 for efficient online monitoring. Since no complete trace is given in online monitoring, $\mathbb{C}\langle X\rangle$ focuses on actions to carry out when a parametric event $e\langle\theta\rangle$ arrives; in other words, it essentially expands the body of the outer loop in $\mathbb{B}\langle X\rangle$ (lines 3 to 7 in Figure 2). We chose not to use $\mathbb{B}\langle X\rangle$ directly for our implementation for efficiency concerns: the inner loop in $\mathbb{B}\langle X\rangle$ requires search for all parameter instances in Θ that are compatible with θ; 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 optimizations to improve the performance of monitoring. While $\mathbb{B}\langle X\rangle$ did not require that θ in $e\langle\theta\rangle$ be of finite domain, $\mathbb{C}\langle X\rangle$ needs that requirement to terminate. Note that in practice $\mathsf{Dom}(\theta)$ is always finite (because the program state is finite).

$\mathbb{C}\langle X\rangle$ uses three tables: $\Delta, \mathcal{U}$ and Γ. Δ and Γ are the same as Δ and Γ 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 Figure 2). It maps each parameter instance θ into the finite set of parameter instances encountered in Δ so far that are strictly more informative than θ, i.e., $\mathcal{U}(\theta) = \{\theta' \mid \theta' \in \mathsf{Dom}(\Delta)$ 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 Θ during computation; instead, Θ is implicitly captured by the domain of Δ in $\mathbb{C}\langle X\rangle$. Intuitively, Θ 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, Θ 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, θ and θ', and adds a new entry corresponding to θ into Δ and $\mathcal{U}$. Specifically, it sets $\Delta(\theta)$ to $\Delta(\theta')$ and adds θ 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 Δ is already defined on θ, i.e., $\theta \in \Theta$ at the beginning of the iteration of the outer loop in $\mathbb{B}\langle X\rangle$. In this case, $\{\theta\} \sqcup \Theta = \{\theta' \mid \theta' \in \Theta$ 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 Δ is not already defined on θ, 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 Δ (lines 2 to 15). We first add an entry to Δ for θ at lines 2 to 7. Then we search for all parameter instances θ_{comp} that are compatible with θ, making use of $\mathcal{U}$ (line 8 and 9); for each such θ_{comp}, an appropriate entry is added to Δ for its lub with θ, and $\mathcal{U}$ updated accordingly (lines 10 to 12). This way, Δ 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 the parameter instance $\max{(\theta]_\Theta}$ that $\mathbb{B}\langle X\rangle$ refers to at line 4 in Figure 2: it enumerates all the $\theta_{max} \sqsubset \theta$ in reversed topological order (from larger to smaller); *1.* in Proposition 8 guarantees that the maximum exists and, since it is unique, our search will find it.

Correctness of $\mathbb{C}\langle X\rangle$**.** We next prove the correctness of $\mathbb{C}\langle X\rangle$ by showing that it is equivalent to the body of the outer loop in $\mathbb{B}\langle X\rangle$. Suppose that parametric trace τ 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.

Let us first note that $\mathbb{C}\langle X\rangle$ terminates if $\mathsf{Dom}(\theta)$ is finite. Indeed, then there is only a finite number of partial maps less informative than θ, 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 $\theta \in [X \overset{\circ}{\to} V_X]$ 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 $O(n \times m)$ to process $e\langle\theta\rangle$, where n is $2^{|\mathsf{Dom}(\theta)|}$ and m is the number of incompatible parameter instances in τ. Parametric properties often have a fixed and small number of parameters, in which case n is not significant. Depending on the trace, m can unavoidably grow arbitrarily large; in the worst case, each event may carry an instance incompatible with the previous ones.

LEMMA 1. *$\mathcal{U}(\theta) = \{\theta' \mid \theta' \in Dom(\Delta)$ and $\theta \sqsubset \theta'\}$ before and after each execution of* defineTo, *for all $\theta \in [X \overset{\circ}{\to} V_X]$.*

Proof. By how $\mathbb{C}\langle X\rangle$ is initialized, for any $\theta \in [X \overset{\circ}{\to} V_X]$ we have $\emptyset = \mathcal{U}(\theta) = \{\theta' \mid \theta' \in \mathsf{Dom}(\Delta)$ and $\theta \sqsubset \theta'\}$ before the first execution of defineTo. Now suppose that $\mathcal{U}(\theta) = \{\theta' \mid \theta' \in \mathsf{Dom}(\Delta)$ and $\theta \sqsubset \theta'\}$ for any $\theta \in [X \overset{\circ}{\to} V_X]$ before an execution of defineTo and show that it also holds after the execution of defineTo. Since defineTo(θ, θ') adds a new parameter instance θ into $\mathsf{Dom}(\Delta)$ and also adds θ into the set $\mathcal{U}(\theta'')$ for any $\theta'' \in [X \overset{\circ}{\to} V_X]$ with $\theta'' \sqsubset \theta$, we still have $\mathcal{U}(\theta) = \{\theta' \mid \theta' \in \mathsf{Dom}(\Delta)$ and $\theta \sqsubset \theta'\}$ for any $\theta \in [X \overset{\circ}{\to} V_X]$ after the execution of defineTo. Also, the only way $\mathbb{C}\langle X\rangle$ adds a new parameter instance θ into $\mathsf{Dom}(\Delta)$ is using defineTo. Therefore the lemma holds. □

Next result establishes the correctness of our implementation. We use the following notation. Recall that we fixed parametric trace τ and event $e\langle\theta\rangle$. Let $\mathcal{U}_{\mathbb{C}}$, $\Delta_{\mathbb{C}}$, and $\Gamma_{\mathbb{C}}$ be the three data-structures maintained by $\mathbb{C}\langle X\rangle(M)$ for some M. Let $\Delta^b_{\mathbb{C}}$ and $\Gamma^b_{\mathbb{C}}$ be the $\Delta_{\mathbb{C}}$ and $\Gamma_{\mathbb{C}}$ when main$(e\langle\theta\rangle)$ begins ("b" stays for "at the beginning"); let $\Delta^e_{\mathbb{C}}$ and $\Gamma^e_{\mathbb{C}}$ be the $\Delta_{\mathbb{C}}$ and $\Gamma_{\mathbb{C}}$ when main$(e\langle\theta\rangle)$ ends ("e" stays for "at the end"; and let $\Delta^m_{\mathbb{C}}$ and $\mathcal{U}^m_{\mathbb{C}}$ be the $\Delta_{\mathbb{C}}$ and $\mathcal{U}_{\mathbb{C}}$ when main$(e\langle\theta\rangle)$ reaches line 16 ("m stays for "in the middle").

THEOREM 3. *The following hold:*

1. *$Dom(\Delta^m_{\mathbb{C}}) = \{\bot, \theta\} \sqcup Dom(\Delta^b_{\mathbb{C}})$;*
2. *$\Delta^m_{\mathbb{C}}(\theta') = \Delta^m_{\mathbb{C}}(\max(\theta']_{Dom(\Delta^b_{\mathbb{C}})})$, for all $\theta' \in Dom(\Delta^m_{\mathbb{C}})$;*
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 $\Gamma^b_{\mathbb{C}} = \mathbb{B}\langle X\rangle(M)(\tau\, e\langle\theta\rangle).\Gamma$.*

Proof. Let $\Theta_{\mathbb{C}} = \mathsf{Dom}(\Delta^b_{\mathbb{C}}) = \mathsf{Dom}(\Delta_{\mathbb{B}}(\tau))$ and $\Delta_{\mathbb{B}}(\tau) = \mathbb{B}\langle X\rangle(M)(\tau\langle\theta\rangle).\Delta$ for simplicity.

1. There are two cases to analyze, depending upon θ is in $\Theta_{\mathbb{C}}$ or not. If $\theta \in \Theta_{\mathbb{C}}$ then lines 2 to 14 are skipped and $\mathsf{Dom}(\Delta_{\mathbb{C}})$ remains unchanged, that is, $\{\bot, \theta\} \sqcup \Theta_{\mathbb{C}} = \Theta_{\mathbb{C}} = \mathsf{Dom}(\Delta^b_{\mathbb{C}}) = \mathsf{Dom}(\Delta^m_{\mathbb{C}})$ when main$(e\langle\theta\rangle)$ reaches line 16. If $\theta \notin \Theta_{\mathbb{C}}$ then lines 2 to 14 are executed to add new parameter instances into $\mathsf{Dom}(\Delta_{\mathbb{C}})$. First, an entry for θ will be added to $\Delta_{\mathbb{C}}$ at line 7. Second, an entry for $\theta_{comp} \sqcup \theta$ will be added to $\Delta_{\mathbb{C}}$ at line 11 (if $\Delta_{\mathbb{C}}$ not already defined on $\theta_{comp} \sqcup \theta$) eventually for any $\theta_{comp} \in \Theta_{\mathbb{C}}$ compatible with θ: that is because θ_{max} can also be $\bot$ at line 8, in which case Lemma 1 implies that $\mathcal{U}(\theta_{max}) = \Theta_{\mathbb{C}}$. Therefore, when line 16 is reached, $\mathsf{Dom}(\Delta^m_{\mathbb{C}})$ is defined on all the parameter instances in $\{\theta\} \cup (\{\theta\} \sqcup \Theta_{\mathbb{C}})$. Since $\bot \in \Theta_{\mathbb{C}}$, the latter equals $\{\theta\} \sqcup \Theta_{\mathbb{C}}$, and since $\Delta^m_{\mathbb{C}}$ remains defined on $\Theta_{\mathbb{C}}$, we conclude that $\Delta^m_{\mathbb{C}}$ is defined on all instances in $(\{\theta\} \sqcup \Theta_{\mathbb{C}}) \cup \Theta_{\mathbb{C}}$, which by *5.* and *7.* in Proposition 4 equals $\{\bot, \theta\} \sqcup \Theta_{\mathbb{C}}$.

2. We analyze the same two cases as above. If $\theta \in \Theta_{\mathbb{C}}$ then lines 2 to 14 are skipped and $\mathsf{Dom}(\Delta_{\mathbb{C}})$ remains unchanged. Then $\max(\theta']_{\Theta_{\mathbb{C}}} = \theta'$ for each $\theta' \in \mathsf{Dom}(\Delta^m_{\mathbb{C}})$, so the result follows. Suppose now that $\theta \notin \Theta_{\mathbb{C}}$. By *1.* and its proof, each $\theta' \in \mathsf{Dom}(\Delta^m_{\mathbb{C}})$ is either in $\Theta_{\mathbb{C}}$ or otherwise in $(\{\theta\} \sqcup \Theta_{\mathbb{C}}) - \Theta_{\mathbb{C}}$. The result immediately holds when $\theta' \in \Theta_{\mathbb{C}}$ as $\max(\theta']_{\Theta_{\mathbb{C}}} = \theta'$ and $\Delta(\theta')$ stays unchanged until line 16. If $\theta' \in (\{\theta\} \sqcup \Theta_{\mathbb{C}}) - \Theta_{\mathbb{C}}$ then $\Delta(\theta')$ is set at either line 7 ($\theta' = \theta$) or at line 11 ($\theta' \neq \theta$):

(a) For line 7, the loop at lines 2 to 6 checks all the parameter instances that are less informative than θ to find the first one in $\Theta_{\mathbb{C}}$ in reversed topological order (i.e., if $\theta_1 \sqsubset \theta_2$ then θ_2 will be checked before θ_1). Since by *1.* in Proposition 8 we know that $\max(\theta]_{\Theta_{\mathbb{C}}} \in \Theta_{\mathbb{C}}$ exists (and it is unique), the loop at lines 2 to 6 will break precisely when $\theta_{max} = \max(\theta]_{\Theta_{\mathbb{C}}}$, so the result holds when $\theta' = \theta$ because of the entry introduced for θ in $\Delta_{\mathbb{C}}$ at line 7 and because the remaining lines 8 to 14 do not change $\Delta_{\mathbb{C}}(\theta)$.

(b) When $\Delta_{\mathbb{C}}(\theta')$ is set at line 11, note that the loop at lines 8 to 14 also iterates over all $\theta_{max} \sqsubset \theta$ in reversed topological order, so $\theta' = \theta_{comp} \sqcup \theta$ for some $\theta_{comp} \in \Theta_{\mathbb{C}}$ compatible with θ such that $\theta_{max} \sqsubset \theta_{comp}$, where $\theta_{max} \sqsubset \theta$ is such that there is no other θ'_{max} with $\theta_{max} \sqsubset \theta'_{max} \sqsubset \theta$ and $\theta' = \theta'_{comp} \sqcup \theta$ for some $\theta'_{comp} \in \Theta_{\mathbb{C}}$ compatible with θ such that $\theta'_{max} \sqsubset \theta'_{comp}$. We claim that there is only one such θ_{comp}, which is precisely $\max(\theta']_{\Theta_{\mathbb{C}}}$: Let θ'_{comp} be the parameter instance $\max(\theta']_{\Theta_{\mathbb{C}}}$. The above implies that $\theta_{comp} \sqsubseteq \theta'_{comp} \sqsubseteq \theta'$. Also, $\theta'_{comp} \sqcup \theta = \theta'$ because $\theta' = \theta_{comp} \sqcup \theta \sqsubseteq \theta'_{comp} \sqcup \theta \sqsubseteq \theta'$. Let θ'_{max} be $\theta'_{comp} \sqcap \theta$, that is, the largest with $\theta'_{max} \sqsubseteq \theta'_{comp}$ and $\theta'_{max} \sqsubseteq \theta$ (we let its existence as exercise). It is relatively easy to see now that $\theta_{comp} \sqsubset \theta'_{comp}$ implies $\theta_{max} \sqsubset \theta'_{max}$ (we let it as an exercise, too), which contradicts the assumption of this case that $\Delta_{\mathbb{C}}$ was not defined on θ'. Therefore, $\theta_{comp} = \max(\theta']_{\Theta_{\mathbb{C}}}$ before line 11 is executed, which means that, after line 11 is executed, $\Delta_{\mathbb{C}}(\theta') = \Delta_{\mathbb{C}}(\max(\theta']_{\Theta_{\mathbb{C}}})$; moreover, none of these will be changed anymore until line 16 is reached, which proves our result.

3. Since Γ is updated according to Δ in both $\mathbb{C}\langle X\rangle$ and $\mathbb{B}\langle X\rangle$, it is enough to prove that $\Delta^e_{\mathbb{C}} = \Delta_{\mathbb{B}}(\tau e)$. For $\mathbb{B}\langle X\rangle$, we have
1) $\mathsf{Dom}(\Delta_{\mathbb{B}}(\tau e)) = \{\bot, \theta\} \sqcup \Theta_{\mathbb{C}} = (\{\theta\} \sqcup \Theta_{\mathbb{C}}) \cup \Theta_{\mathbb{C}}$;
2) $\forall\, \theta' \in \{\theta\} \sqcup \Theta_{\mathbb{C}}$, $\Delta_{\mathbb{B}}(\tau e)(\theta') = \sigma(\Delta_{\mathbb{B}}(\tau)(\max, (\theta']_{\Theta_{\mathbb{C}}}), e)$;
3) $\forall\, \theta' \in \Theta_{\mathbb{C}} - \{\theta\} \sqcup \Theta_{\mathbb{C}}$, $\Delta_{\mathbb{B}}(\tau e)(\theta') = \Delta_{\mathbb{B}}(\tau)(\theta')$.
So we only need to prove that
1) $\mathsf{Dom}(\Delta^e_{\mathbb{C}}) = \{\bot, \theta\} \sqcup \Theta_{\mathbb{C}}$;
2) $\forall\, \theta' \in \{\theta\} \sqcup \Theta_{\mathbb{C}}$, $\Delta^e_{\mathbb{C}}(\theta') = \sigma(\Delta^b_{\mathbb{C}}(\max, (\theta']_{\Theta_{\mathbb{C}}}), e)$;
3) $\forall\, \theta' \in \Theta_{\mathbb{C}} - \{\theta\} \sqcup \Theta_{\mathbb{C}}$, $\Delta^e_{\mathbb{C}}(\theta') = \Delta^b_{\mathbb{C}}(\theta')$.
By *1.*, we have $\mathsf{Dom}(\Delta^m_{\mathbb{C}}) = \{\bot, \theta\} \sqcup \Theta_{\mathbb{C}}$. Since lines 16 to 19 do not change $\mathsf{Dom}(\Delta_{\mathbb{C}})$, $\mathsf{Dom}(\Delta^e_{\mathbb{C}}) = \mathsf{Dom}(\Delta^m_{\mathbb{C}}) = \{\bot, \theta\} \sqcup \Theta_{\mathbb{C}}$. 1) holds.

By *2.* and Lemma 1, $\Delta^m_{\mathbb{C}}(\theta') = \Delta^b_{\mathbb{C}}(\max, (\theta']_{\Theta_{\mathbb{C}}})$ for any $\theta' \in \mathsf{Dom}(\Delta^m_{\mathbb{C}})$. Also, notice that line 17 sets $\Delta_{\mathbb{C}}(\theta')$ to $\sigma(\Delta_{\mathbb{C}}(\theta'), e)$, which is $\sigma(\Delta^b_{\mathbb{C}}(\max, (\theta']_{\Theta_{\mathbb{C}}}), e)$, for the θ' in the loop. So, to show 2) and 3), we only need to prove that the loop at line 16 to 19 iterates over $\{\theta\} \sqcup \Theta_{\mathbb{C}}$. Since lines 16 to 19 do not change $\mathcal{U}_{\mathbb{C}}$, we need to show $\{\theta\} \cup \mathcal{U}^m_{\mathbb{C}}(\theta) = \{\theta\} \sqcup \Theta_{\mathbb{C}}$. Since $\mathsf{Dom}(\Delta^m_{\mathbb{C}}) = \{\bot, \theta\} \sqcup \Theta_{\mathbb{C}}$, we have $\{\theta\} \sqcup \mathsf{Dom}(\Delta^m_{\mathbb{C}}) = \{\theta\} \sqcup (\{\bot, \theta\} \sqcup \Theta_{\mathbb{C}}) = \{\theta\} \sqcup ((\{\theta\} \sqcup \Theta_{\mathbb{C}}) \cup \Theta_{\mathbb{C}})$. By Proposition 4, $\{\theta\} \sqcup \mathsf{Dom}(\Delta^m_{\mathbb{C}}) = (\{\theta\} \sqcup (\{\theta\} \sqcup \Theta_{\mathbb{C}})) \cup (\{\theta\} \sqcup \Theta_{\mathbb{C}}) = (\{\theta\} \sqcup \Theta_{\mathbb{C}}) \cup (\{\theta\} \sqcup \Theta_{\mathbb{C}}) = \{\theta\} \sqcup \Theta_{\mathbb{C}}$. Also, as $\theta \in \mathsf{Dom}(\Delta^m_{\mathbb{C}})$, we have $\{\theta\} \sqcup \mathsf{Dom}(\Delta^m_{\mathbb{C}}) = \{\theta' \mid \theta' \in \mathsf{Dom}(\Delta^m_{\mathbb{C}})$ and $\theta \sqsubseteq \theta'\} = \{\theta\} \sqcup \mathcal{U}^m_{\mathbb{C}}(\theta)$ by Lemma 1. So $\{\theta\} \cup \mathcal{U}^m_{\mathbb{C}}(\theta) = \{\theta\} \sqcup \Theta_{\mathbb{C}}$. □

Optimizations. We next discuss some optimizations implemented in our prototype tool. First, $\mathbb{C}\langle X\rangle$ iterates through all the possible parameter instances that are less informative than θ in three different loops: at lines 2 and 8 in the main function, and at line 2 in the defineTo function. Hence, it is important to reduce the number of such instances in each loop. When the number of parameters is finite, a simple static analysis of the events appearing in a specification allows us to quickly detect parameter instances that can never appear as lubs of instances of parameters carried by events; maintaining any space for those in Δ, or Γ, or iterating over them in the above mentioned loops, is a waste. For example, if a specification contains only two event definitions, $e_1\langle p_1\rangle$ and $e_2\langle p_1, p_2\rangle$, parameter instances defining only parameter p_2 can never appear as lubs of observed parameter instances. A more advanced static analysis of the specification, discussed in [8], exhaustively explores all possible event combinations that can lead to situations of interest to the property, such as to violation, validation, etc. Such information can be used to reduce the number of loop iterations by skipping iterations over parameter instances that cannot affect the result of monitoring. These static analyses can therefore be used at compile time to unroll the loops in $\mathbb{C}\langle X\rangle$ and reduce the size of Δ and $\mathcal{U}$.

Another optimization is based on the observation that it is common to start the monitoring process 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 implementation of JavaMOP [9] supports only traces whose monitoring starts with a fully instantiated monitor creation event; this was perceived (and admitted) as a limitation of JavaMOP [3, 9] (a performance trade-off). Interestingly, it now becomes just an optimization of our general and unrestricted technique for a very common case.

Experiments and Evaluation. We have implemented our online monitoring algorithm above together with the discussed optimizations in a prototype, here called PMon (from ”Parametric Monitoring”), and evaluated it on the DaCapo benchmark [6]. The following properties from [8] were checked in our experiments:

LeakingSync. Only access a synchronized collection using its synchronized wrapper. One violation pattern is monitored:

$$\Lambda c.\ \mathsf{sync}(c)\ \mathsf{asyncAccess}(c)$$

ASyncIterCollection. Only iterate a synchronized collection c when holding a lock on c. Two violation patterns are monitored:

$$\Lambda c, i.\ \mathsf{sync}(c)\ \mathsf{ayncCreateIter}(c, i)$$

$$\Lambda c, i.\ \mathsf{sync}(c)\ \mathsf{syncCreateIter}(c, i)\ \mathsf{asyncAccess}(i)$$

ASyncIterMap. Only iterate a synchronized map m when holding a lock on m. Two violation patterns are monitored:

$$\Lambda m, s, i.\ \mathsf{sync}(m)\ \mathsf{getSet}(m, s)\ \mathsf{asyncCreateIter}(s, i)$$

$$\Lambda m, s, i.\ \mathsf{sync}(m)\ \mathsf{getSet}(m, s)\ \mathsf{syncCreateIter}(s, i)\ \mathsf{asyncAccess}(i)$$

FailSafeEnum. Do not update a vector while enumerating over it. The following violation pattern is monitored:

$$\Lambda v, e.\ \mathsf{createEnum}(v, e)\ \mathsf{modify}(v)\ \mathsf{access}(e)$$

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

Using the above properties, we compared our implementation with three other monitoring approaches, namely: manually implemented monitoring[2], Tracematches and JavaMOP. We chose these systems for comparison because they are very efficient runtime verification systems [3, 9]. All the experiments were carried out on a 1.5GB RAM Pentium 4 2.66GHz processor running Ubuntu Linux 7.10. We used the DaCapo benchmark version 2006-10; it contains eleven open source programs: antlr, bloat, chart, eclipse, fop, hsqldb, jython, luindex, lusearch, pmd, and xalan. 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. 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.

[2] The manually implemented monitoring code was borrowed from [8].

10. 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. A parametric monitoring technique, also discussed and proved correct, makes it possible 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 ad hoc and limited, techniques in current use, does not come at a performance expense.

The parametric trace slicing technique in Section 6 enables the leveraging of any non-parametric, i.e., conventional, trace analysis techniques to the parametric case. We have only considered monitoring in this paper. Another interesting and potentially rewarding use of our technique could be in the context of property mining. For example, one could run the trace slicing algorithm on large benchmarks making intensive use of library classes, and then, on the obtained trace slices corresponding to particular classes or groups of classes of interest, run property mining algorithms. The mined properties, or the lack thereof, may provide insightful formal documentation for libraries, or even detect errors. We plan to incorporate this and the parametric monitoring algorithm in JavaMOP.

References

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA’05*, 2005.

[2] AspectJ. http://eclipse.org/aspectj/.

[3] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitoring feasible. In R. P. Gabriel, editor, *ACM Conference on Object-Oriented Programming, Systems and Languages (OOPSLA’07)*. ACM Press, 2007.

[4] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In B. Steffen and G. Levi, editors, *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004.

[5] H. Barringer, D. E. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: From eagleto ruler. In O. Sokolsky and S. Tasiran, editors, *RV*, volume 4839 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2007.

[6] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA’06*, pages 169–190. ACM, 2006.

[7] E. Bodden. J-lo, a tool for runtime-checking temporal assertions. Master’s thesis, RWTH Aachen University, 2005.

[8] E. Bodden, F. Chen, and G. Roşu. Dependent advice: A general approach to optimizing history-based aspects (Extended version). Technical Report abc-2008-2, http://www.aspectbench.org/, March 2008.

	LeakingSync				ASyncIterCollection				ASyncIterMap				FailSafeEnum			
	PMon	Manual	TM	MOP	PMon	Manual	TM	MOP	PMon	Manual	TM	MOP	PMon	Manual	TM	MOP
antlr	2	1	5	2	1	0	0	n/a	2	1	2	n/a	0	0	1	0
bloat	140	145	785	141	149	150	1459	n/a	179	164	2300	n/a	0	2	0	2
chart	25	21	70	24	2	0	0	n/a	0	0	0	n/a	1	0	0	0
eclipse	0	0	0	0	1	2	0	n/a	1	0	0	n/a	0	0	0	2
fop	53	47	146	50	2	1	1	n/a	2	1	0	n/a	1	0	0	0
hsqldb	2	5	24	4	1	0	25	n/a	1	0	0	n/a	0	0	25	0
jython	62	52	55	59	0	0	9	n/a	0	0	9	n/a	0	0	8	0
luindex	8	7	20	7	3	0	2	n/a	1	0	4	n/a	7	4	16	3
lusearch	12	10	52	12	4	1	9	n/a	0	0	8	n/a	5	2	28	7
pmd	55	47	53	52	37	30	36	n/a	50	49	53	n/a	0	0	0	0
xalan	39	29	117	40	1	1	6	n/a	1	1	7	n/a	5	6	33	4

Table 2. Average percent runtime overhead for PMon, manually coded monitors, Tracematches and JavaMOP (convergence within 3%).

[9] F. Chen and G. Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'07)*, pages 569–588. ACM Press, 2007.

[10] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[11] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *OOPSLA'05*, 2005.

[12] S. Maoz and D. Harel. From multi-modal scenarios to code: compiling lscs into aspectj. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 219–230, 2006.

[13] M. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA'05*, 2005.

[14] P. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering(ASE '08)*. IEEE/ACM, 2008. to appear 2008.

[15] E. F. Moore. Gedanken-experiments on sequential machines. *Automata Studies, Annals of Mathematical Studies*, 34:129–153, 1956.

[16] G. Roşu, F. Chen, and T. Ball. Synthesizing monitors for safety properties – this time with calls and returns –. In *Workshop on Runtime Verification (RV'08)*, 2008. to appear.

[17] G. Rosu and K. Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 12(2):151–197, 2005.