SEMANTICS AND ALGORITHMS FOR PARAMETRIC MONITORING

GRIGORE ROŞU AND FENG CHEN

Department of Computer Science, University of Illinois at Urbana-Champaign
e-mail address: grosu@illinois.edu

ABSTRACT. Analysis of execution traces plays a fundamental role in many program analysis approaches, such as runtime verification, testing, monitoring, and specification mining. Execution traces are frequently parametric, i.e., they contain events with parameter bindings. Each parametric trace usually consists of many meaningful *trace slices* merged together, each slice corresponding to one parameter binding. For example, a Java program creating iterator objects i_1 and i_2 over collection object c_1 may yield a trace createIter$\langle c_1 \, i_1 \rangle$ next$\langle i_1 \rangle$ createIter$\langle c_1 \, i_2 \rangle$ updateColl$\langle c_1 \rangle$ next$\langle i_1 \rangle$ parametric in collection c and iterator i, whose slices corresponding to instances "$c, i \mapsto c_1, i_1$" and "$c, i \mapsto c_1, i_2$" are createIter$\langle c_1 \, i_1 \rangle$ next$\langle i_1 \rangle$ updateColl$\langle c_1 \rangle$ next$\langle i_1 \rangle$ and, respectively, createIter$\langle c_1 \, i_2 \rangle$ updateColl$\langle c_1 \rangle$. Several approaches have been proposed to specify and dynamically analyze parametric properties, but they have limitations: some in the specification formalism, others in the type of trace they support. Not unexpectedly, the existing approaches share common notions, intuitions, and even techniques and algorithms, suggesting that a fundamental study and understanding of parametric trace analysis is necessary.

This foundational paper aims at giving a semantics-based solution to parametric trace analysis that is unrestricted by the type of parametric property or trace that can be analyzed. Our approach is based on a rigorous understanding of *what* a parametric trace/property/monitor is and *how* it relates to its non-parametric counter-part. A general-purpose parametric trace slicing technique is introduced, which takes each event in the parametric trace and dispatches 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.
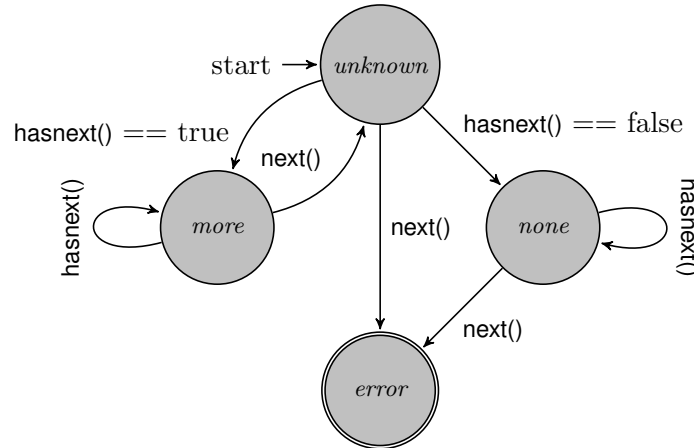
GRIGORE ROŞU AND FENG CHEN

Figure 1: Typestate property describing the correct use of the next() and hasnext() methods.

1. Introduction and Motivation

Parametric traces, i.e., traces containing events with parameter bindings, abound in program executions, because they naturally appear whenever abstract parameters (e.g., variable names) are bound to concrete data (e.g., heap objects) at runtime. In this section we first discuss some motivating examples and describe the problem addressed in this paper, then we recall related work and put the work in this paper in context, and then we explain our contributions and finally the structure of the paper.

1.1. **Motivating examples and highlights.** We here describe three examples of parametric properties, in increasing difficulty order, and use them to highlight and motivate the semantic results and the algorithms presented in the rest of the paper.

Typestates [34] refine the notion of type by stating not only what operations are allowed by a particular object, but also what operations are allowed in what contexts. Typestates are particular parametric properties with only one parameter. Figure 1 shows the typestate description for a property saying that it is invalid to call the next() method on an iterator object when there are no more elements in the underlying collection, i.e., when hasnext() returns false, or when it is unknown if there are more elements in the collection, i.e., hasnext() is not called. From the *unknown* state, it is always an error to call the next() method because such an operation could be unsafe. If hasnext() is called and returns true, it is safe to call next(), so the typestate enters the *more* state. If, however, the hasnext() method returns false, there are no more elements, and the typestate enters the *none* state. In the *more* and *none* states, calling the hasnext() method provides no new information. It is safe to call next() from the *more* state, but it becomes unknown if more elements exist, so the typestate reenters the initial *unknown* state. Finally, calling next() from the *none* state results in an error. For simplicity, we here assume that next() is the only means to modify the state of an iterator; concurrent modifications are discussed in other examples shortly.

It is straightforward to represent the typestate property in Figure 1, and all typestate properties, as particular (one-parameter) *parametric* properties. Indeed, the behaviors described by the typestate in Figure 1 are intended to be obeyed by all iterator object instances; that is, we have a property parametric in the iterator. To make this more precise,

let us look at the problem from the perspective of observable program *execution traces*. A trace can be regarded as a sequence of *events* relevant to the property of interest, in our case calls to next() or to hasnext(); the latter can be further split into two categories, one when hasnext() returns true and the other when it returns false. Since the individuality of each iterator matters, we must regard each event as being *parametric* in the iterator yielding it. Formally, traces relevant to our typestate property are formed with three parametric events, namely next$\langle i \rangle$, hasnexttrue$\langle i \rangle$, and hasnextfalse$\langle i \rangle$. A possible trace can be hasnexttrue$\langle i_1 \rangle$ hasnextfalse$\langle i_2 \rangle$ next$\langle i_1 \rangle$ next$\langle i_2 \rangle$..., which violates the typestate property for iterator instance i_2. How to obtain execution traces is not our concern here (several runtime monitoring systems use AspectJ instrumentation). Our results in this paper are concerned with how to specify properties over parametric traces, what is their meaning, and how to monitor them.

Let us first briefly discuss our approach to specifying properties over parametric execution traces, that is, *parametric properties*. To keep it as easy as possible for the user and to leverage our knowledge on specifying ordinary, non-parametric properties, we build our specification approach on top of *any* formalism for specifying non-parametric properties. More precisely, all one has to do is to first specify the property using any conventional formalism as if there were only one possible instance of its parameters, and then use a special Λ quantifier to make it parametric. For our typestate example, suppose that typestate is the finite state machine in Figure 1, modified by replacing the method calls on edges with actual events as described above. Then the desired parametric property is Λi . typestate. The meaning of this parametric property is that whatever was intended for its non-parametric counterpart, typestate, must hold *for each* parameter instance; that is, the *error* state must not be reached for any iterator instance, which is precisely the desired meaning of this typestate. Another way to specify the same property is using a regular expression matching all the good behaviors, each bad prefix for any instance thus signaling a violation:

$$\Lambda i \, . \, (\text{hasnexttrue}\langle i \rangle^+ \text{next}\langle i \rangle \mid \text{hasnextfalse}\langle i \rangle^*)^*$$

Yet another way to specify its non-parametric part is with a linear-temporal logic formula:

$$\Lambda i \, . \, \square(\text{next}\langle i \rangle \implies \odot \text{hasnexttrue}\langle i \rangle)$$

The above LTL formula says "it is always ($\square$) the case that each next$\langle i \rangle$ event is preceded ($\odot$) by a hasnexttrue$\langle i \rangle$. The LTL formula must hold for any iterator instance. In general, if $\Lambda X \, . \, P$ is a parametric property, where X is a set of one or more parameters, we may call P its corresponding *base* or *root* or *non-parametric property*. In this paper we develop a mathematical foundation for specifying such parametric properties independently of the formalism used for specifying their non-parametric part, define their precise semantics, provide algorithms for online monitoring of parametric properties, and finally bring empirical evidence showing that monitoring parametric properties is in fact feasible.

Parametric properties properly *generalize* typestates in two different directions. First, parametric properties allow more than one parameter, allowing us to specify not only properties about a given object such as the typestate example above, but also properties that capture *relationships between objects*. Second, they allow us to specify infinite-state root properties using formalisms like context-free grammars (see Sections 2.4, 2.5 and 2.6).

Let us now consider a two-parameter property. Suppose that one is interested in analyzing collections and iterators in Java. Then execution traces of interest may contain events createlter$\langle c \, i \rangle$ (iterator i is created for collections c), updateColl$\langle c \rangle$ (c is modified), and next$\langle i \rangle$ (i is accessed using its next element method), instantiated for particular collection and iterator instances. Most properties of parametric traces are also parametric; for our

4   GRIGORE ROȘU AND FENG CHEN

example, a property may be "collections are not allowed to change while accessed through
iterators", which is parametric in a collection *and* an iterator. The parametric property
above expressed as a regular expression (here matches mean violations) can be

$$\Lambda c, i \,.\ \mathsf{createIter}\langle c\,i\rangle\ \mathsf{next}\langle i\rangle^*\ \mathsf{updateColl}\langle c\rangle^+\ \mathsf{next}\langle i\rangle$$

From here on, when we know the number and types of parameters of each event, we omit
writing them in parametric properties, because they are redundant; for example, we write

$$\Lambda c, i \,.\ \mathsf{createIter}\ \mathsf{next}^*\ \mathsf{updateColl}^+\ \mathsf{next}$$

Parametric properties, unfortunately, are very hard to formally verify and validate
against real systems, mainly because of their dynamic nature and potentially huge or even
unlimited number of parameter bindings. Let us extend the above example: in Java, one
may create a collection from a map and use the collection's iterator to operate on the map's
elements. A similar safety property is: "maps are not allowed to change while accessed
indirectly through iterators". Its violation pattern is:

$$\Lambda m, c, i \,.\ \mathsf{createColl}\ (\mathsf{updateMap}\mid\mathsf{updateColl})^*\ \mathsf{createIter}\ \mathsf{next}^*(\mathsf{updateMap}\mid\mathsf{updateColl})^+\ \mathsf{next}$$

with two new parametric events $\mathsf{createColl}\langle m\,c\rangle$ (collection c is created from map m) and
$\mathsf{updateMap}\langle m\rangle$ (m is updated). All the events used in this property provide only partial
parameter bindings ($\mathsf{createColl}$ binds only m and c, etc.), and parameter bindings carried by
different events may be combined into larger bindings; e.g., $\mathsf{createColl}\langle m_1\,c_1\rangle$ can be combined
with $\mathsf{createIter}\langle c_1\,i_1\rangle$ into a full binding $\langle m_1\,c_1\,i_1\rangle$, and also with $\mathsf{createIter}\langle c_1\,i_2\rangle$ into $\langle m_1\,c_1\,i_2\rangle$.
It is highly challenging for a trace analysis technique to correctly and efficiently maintain,
locate and combine trace slices for different parameter bindings, especially when the trace
is long and the number of parameter bindings is large.

This paper addresses the problem of parametric trace analysis from a foundational,
semantic 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 one of the central
results in parametric trace analysis. In short, our overall approach to monitor a parametric
property $\Lambda X\,.\,P$ is to observe the parametric trace as it is being generated by the running
system, slice it online with respect to the used parameter instances, and then send each
slice piece-wise to a non-parametric monitor corresponding to the base property P; this
way, multiple monitor instances for P can and typically do coexist, one for each trace slice.

The main conceptual limitation of our approach is that the parametrization of properties
is only allowed at the top-level, that is, the base property P in the parametric property
$\Lambda X\,.\,P$ cannot have any Λ binders. In other words, we do not consider nested parameters.
To allow nested parameters one needs a syntax for properties, so that one can incorporate
the syntax for parameters within the syntax for properties. However, one of our major goals
is to be formalism-independent, which means that, by the nature of the problem that we
are attempting to solve, we can only parameterize properties at the top. Many runtime
verification approaches deliberately accept the same limitation, as discussed below, because

arbitrarily nested parameters are harder to understand and turn out to generate higher runtime overhead in the systems supporting them.

Our concrete contributions are explained after the related work.

1.2. **Related Work.** We here discuss several major approaches that have been proposed so far to specify and monitor parametric properties, and relate them to our subsequent results in this paper. It is worth mentioning upfront that, except for the MOP approach [28] which motivated and inspired the work in this paper, the existing approaches do *not* follow the general methodology proposed by our approach in this paper. More precisely, they employ a monolithic monitoring approach, where one monitor is associated to each parametric property; the monitor receives and handles each parametric event in a formalism-specific way. In contrast, our approach is to generate multiple local monitors, each keeping track of one parameter instance. Our approach leads not only to a lower runtime overhead as empirically shown in Section 9, but it also allows us to separate concerns (i.e., the parameter handling from the specification formalism and monitor synthesis for the basic property) and thus potentially enabling a broader spectrum of optimizations that work for various different property specification formalisms and corresponding monitors.

Tracematches [1, 4] is an extension of AspectJ [2] supporting parametric regular patterns; when patterns are matched during the execution, user-defined advice can be triggered. J-LO [9] is a variation of Tracematches that supports a first-order extension variant of linear temporal logic (LTL) that supports data parametrization by means of quantifiers [33]; the user-provided actions are executed when the LTL properties are violated. Also based on AspectJ, [24] proposes Live Sequence Charts (LSC) [17] as an inter-object scenario-based specification formalism; LSC is implicitly parametric, requiring parameter bindings at runtime. Tracematches, J-LO and LSC [24] support a limited number of parameters, and each has its own approach to handle parameters, specific to its particular specification formalism. Our semantics-based approach in this paper is generic in the specification formalism and admits, in theory, a potentially unlimited number of parameters. In spite of the generality of our theoretical results, we chose in our current implementations (see Section 9) to also support only a bounded number of parameters, like in the aforementioned approaches.

JavaMOP [28, 14] (`http://javamop.org`) is a parametric specification and monitoring system 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. Until recently, JavaMOP's genericity came at a price: it could only monitor execution traces in which the first event in each slice instantiated all the property parameters. This limitation prevented the JavaMOP system presented in [14] from monitoring some basic parametric properties, including ones discussed in this paper. Our novel approach to parametric trace slicing and monitoring discussed in this paper does not have that limitation anymore. The parametric slicing and monitoring technique discussed in this paper has been incorporated both in JavaMOP [28] and in its commercial-grade successor RV [27], together with several optimizations that we do not discuss here; Section 9 discusses experiments done with both these systems, as well as with Tracematches, for comparison, because Tracematches has proven to be the most efficient runtime verification system besides JavaMOP.

Program Query Language (PQL) [25] allows the specification and monitoring of parametric context-free grammar (CFG) patterns. Unlike the approaches above that only allow

a bounded 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 not 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 (but as mentioned above, our current implementation supports only a bounded number of parameters). Unlike PQL, our semantics and techniques are not limited to particular events, and are generic in the property specification formalism; CFGs are just one such possible formalism.

Eagle [5], RuleR [6], and Program Trace Query Language (PTQL) [19] 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 [4, 26, 14] (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. 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 $\Lambda X . P$.

1.3. **Contributions.** Besides proposing a formal semantics to parametric traces, properties, and monitoring, we make two theoretical contributions and discuss implementations that validate them empirically:

(1) Our first result is a general-purpose online parametric trace slicing algorithm (algorithm $\mathbb{A}\langle X \rangle$ in Section 6) together with its proof of correctness (Theorem 1), positively answering the following question: *given a parametric execution trace, can one effectively find the slices corresponding to each parameter instance without having to traverse the trace for each instance?*

(2) Our second result, building upon the slicing algorithm, is an online monitoring technique (algorithms $\mathbb{B}\langle X \rangle$ and $\mathbb{C}\langle X \rangle$ in Section 8) together with its proof of correctness (Theorems 2 and 3), which positively answers the following question: *is it possible to monitor arbitrary parametric properties $\Lambda X . P$ against parametric traces, provided that the root property P is monitorable using conventional monitors?*

(3) Finally, our implementation of these techniques in the JavaMOP and RV systems positively answers the following question: *can we implement general purpose and unrestricted parametric property monitoring tools which are comparable in performance with or even outperform existing parametric property monitoring tools on the restricted types of properties and/or traces that the latter support?*

Preliminary results reported in this paper have been published in a less polished form as a technical report in summer 2008 [30]. Then a shorter, conference paper was presented at TACAS 2009 in York, U.K. [16]. This extended paper differs from [16] as follows:

(1) It defines all the mathematical infrastructure needed to prove the results claimed in [16]. For example, Section 3 is new.

(2) It expands the results in [16] and includes all their proofs, as well as additional results needed for those proofs. For example, Section 5 is new.

(3) It discusses more examples of parametric properties. For example, Section 2 is new.

(4) The implementation section in [16] presented an incipient implementation of our technique in a prototype system called PMon there (from Parametric Monitoring). In the meanwhile, we have implemented the technique described in this paper as an integral part of the runtime verification systems JavaMOP (`http://javamop.org`) and RV [27]. The implementation section (Section9) now refers to these systems.

1.4. **Paper Structure.** Section 2 discusses examples of parametric properties. Section 3 provides the mathematical background needed to formalize the concepts introduced later in the paper. Section 4 formalizes parametric events, traces and properties, and defines trace slicing. Section 5 establishes a tight connection between the parameter instances in a trace and the parameter instance used for slicing. Sections 6, 7 and 8 discuss our main techniques for parametric trace slicing and monitoring, and prove them correct. Section 9 discusses implementations of these techniques in two related systems, JavaMOP and RV. Section 10 concludes and proposes future work.

2. Examples of Parametric Properties

In this section we discuss several examples of parametric properties. Our purpose here is twofold. On the one hand we give the reader additional intuition and motivation for the subsequent semantics and algorithms, and, on the other hand, we justify the generality of our approach with respect to the employed specification formalism for trace properties. The discussed examples of parametric properties are defined using various trace specification formalisms, some with more than one parameter and some with more than validating and/or violating categories of behaviors. For each of the examples, we give hints on how our subsequent techniques in Sections 6 and 8 work. In order to explain the examples in this section we also informally introduce necessary notions, such as events and traces (both parametric and non-parametric); all these notions will be formally defined in Section 4.

For each example, we also discuss which of the existing runtime verification systems can support it. Note that JavaMOP [28] and its commercial-grade successor RV [27], which build upon the trace slicing and monitoring techniques presented in this paper, are the only runtime verification systems that support all the parametric properties discussed below.

2.1. **Releasing acquired resources.** Consider a certain type of resource (e.g., synchronization objects) that can be acquired and released by a given procedure, and suppose that we want the resources of this type to always be explicitly released by the procedure whenever acquired and only then. This example will be broken in subparts and used as a running example in Section 4 to introduce our main notions and notations.

Let us first consider the non-parametric case in which we have only one resource. Supposing that the four events of interest, i.e., the begin/end of the procedure and the acquire/release of the resource, are $\mathcal{E} = \{\mathsf{begin}, \mathsf{end}, \mathsf{acquire}, \mathsf{release}\}$, then the following regular pattern P captures the desired behavior requirements:

$$P = (\mathsf{begin}(\epsilon \mid (\mathsf{acquire}(\mathsf{acquire} \mid \mathsf{release})^* \mathsf{release}))\mathsf{end})^*$$

The above regular pattern states that the procedure can take place multiple times and, if the resource is acquired then it is released by the end of the procedure (ϵ is the empty word). For simplicity, we here assume that the procedure is not recursive and that the resource can be acquired and released multiple times, with the effect of acquiring and respectively releasing it precisely once; Section 2.4 shows how to use a context-free pattern to specify possibly recursive procedures with matched acquire/release events within each procedure invocation. One matching execution trace for this property is, e.g., begin acquire acquire release end begin end.

Let us now consider the parametric case in which we may have more than one resource and we want each of them to obey the requirements specified above. Now the events acquire and release are parametric in the resource being acquired or released, that is, they have the form acquire$\langle r_1 \rangle$, release$\langle r_2 \rangle$, etc. The begin/end events take no parameters, so we write them begin$\langle \rangle$ and end$\langle \rangle$. A parametric trace τ for our running example can be the following:

$$\tau = \text{begin}\langle \rangle \; \text{acquire}\langle r_1 \rangle \; \text{acquire}\langle r_2 \rangle \; \text{acquire}\langle r_1 \rangle \; \text{release}\langle r_1 \rangle \; \text{end}\langle \rangle \; \text{begin}\langle \rangle \; \text{acquire}\langle r_2 \rangle \; \text{release}\langle r_2 \rangle \; \text{end}\langle \rangle$$

This trace involves two resources, r_1 and r_2, and it really consists of *two trace slices* merged together, one for each resource:

$$\langle r_1 \rangle : \quad \text{begin acquire acquire release end begin end}$$
$$\langle r_2 \rangle : \quad \text{begin acquire end begin acquire release end}$$

The begin and end events belong to both trace slices. Since we know the parameter instance for each trace slice and we know the types of parameters for each event, to avoid clutter we do not mention the redundant parameter bindings of events in trace slices.

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 a collection of meaningful trace slices so that it can quickly identify and report the slice corresponding to any parameter instance when requested.

Note that the $\langle r_1 \rangle$ trace slice matches the specification P above, while the $\langle r_2 \rangle$ trace slice does not. To distinguish parametric properties referring to multiple trace slices from ordinary properties, we explicitly list the parameters using a special Λ binder. For example, our property above parametric in the resource r is $\Lambda r \, . \, P$, or

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

Both Tracematches [1, 4] and JavaMOP [28] can specify/monitor such parametric regular properties, the latter using its extended-regular expression (ERE) plugin.

For the sake of a terminology, P is called a non-parametric, or a root, or a basic property, in contrast to $\Lambda r \, . \, P$, which is called a parametric property. As detailed in Section 4, parametric properties are functions taking a parametric trace (e.g., τ) and a parameter instance (e.g., $r \mapsto r_1$ or $r \mapsto r_2$) into a verdict category for the basic property P (e.g., match or fail). In our case, the semantics of our parametric property $\Lambda r \, . \, P$ takes parametric trace τ and parameter instance $r \mapsto r_1$ to match, and takes τ and $r \mapsto r_2$ to fail, that is,

$$(\Lambda r \, . \, P)(\tau)(r \mapsto r_1) = \text{match}$$
$$(\Lambda r \, . \, P)(\tau)(r \mapsto r_2) = \text{fail}$$

Our parametric monitoring algorithm in Section 8 reports a fail for instance $r \mapsto r_2$ precisely when the first end event is encountered.

We would like to make two observations at this stage. First, as we already mentioned, we only parameterize a property at the top, that is, the Λ binder cannot be used inside the basic property. Indeed, since we do not enforce any particular syntax for basic properties,

it is not clear how to mix the Λ binder with the inexistent property constructs. Second, one should not confuse our parameters with universally quantified variables. While in our example above Λ may feel like a universal quantifier, note that one may prefer to specify the same parametric property in a more negative fashion, for example to specify the bad behaviors instead of the positive ones. Relying on the fact that the begin and end events must be correctly matched, one can only state the bad patterns, which are a begin followed by a release and an acquire followed by an end:

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

The right way to regard a parametric property is as one indexed by all possible instances of the parameters, each instance having its own interpretation of the trace (only caring of the events relevant to it), which is orthogonal to the other instances' interpretations.

2.2. **Authenticate before use.** Consider a server authenticating and using keys, say k_1, k_2, k_3, etc., whose execution traces contain events $\mathsf{authenticate}\langle k_1 \rangle$, $\mathsf{use}\langle k_2 \rangle$, 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 (LTL) as a specification formalism for the corresponding base property, can be expressed as

$$\Lambda k \, . \, \square(\mathsf{use} \rightarrow \diamond \mathsf{authenticate})$$

Such parametric LTL properties can be expressed in both J-LO [9] and JavaMOP [28, 14] (the later using its LTL logic plugin). For the trace above, the trace slice corresponding to k_3 is 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 k_3 in τ, but keeping only the base events; also, the trace slice corresponding to k_2 is use authenticate use. Our trace slicing algorithmin Section 6 can detect these slices. Moreover, with the finite trace LTL semantics in [32],

$$(\Lambda k \, . \, \square(\mathsf{use} \rightarrow \diamond \mathsf{authenticate}))(\tau)(k \mapsto k_3) = \mathsf{true}$$
$$(\Lambda k \, . \, \square(\mathsf{use} \rightarrow \diamond \mathsf{authenticate}))(\tau)(k \mapsto k_2) = \mathsf{false}$$

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.

2.3. **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. The JVM usually throws a runtime exception when this occurs, but the exception is not guaranteed in a multi-threaded environment. 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 when 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 property

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

Such parametric regular expression properties can be expressed in both Tracematches [1] and JavaMOP [28, 14] (the latter using its ERE plugin). We here assumed that the matching of the regular expression corresponds to violation of the base property. 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 the slice corresponding to $\langle v_1 \, i_1 \rangle$ is $\mathsf{create\ next\ update\ next}$ and the one corresponding to $\langle v_1 \, i_2 \rangle$ is $\mathsf{create\ update}$, so τ violates the parametric property (i.e., matches the regular pattern above) on instance $\langle v_1 \, i_1 \rangle$, but not on instance $\langle v_1 \, i_2 \rangle$. Note that in this example there are more than one 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 $\langle v_1 \, i_2 \rangle$, it can do so without any further analysis of the trace. Also, in this case, the monitoring algorithm in Section 8 reports a match each time a parameter instance yields a matching trace slice.

2.4. **Correct locking.** Consider a custom implementation of synchronization in which one can acquire and release locks manually (like in Java 5 and later versions). 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 [25] and JavaMOP [28, 26] (the later using its CFG plugin). We here borrow the CFG property semantics of the CFG plugin of JavaMOP (and also RV [27]) in [26], 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 grammar language. While this property can be expressed in JavaMOP and even monitored in its non-parametric form, the previous implementation of JavaMOP in [26] 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 previous limitation of JavaMOP (allowing only events that instantiate all property's parameters to create a monitor instance) did not allow us to monitor this natural CFG property. To circumvent this limitation, [26] proposed a different way to specify this property, in which the violating traces started with an $\mathsf{acquire}\langle l \rangle$ event. We do not need such artificial encodings anymore in the new version of JavaMOP, and they were never needed in RV (RV improves the new JavaMOP).

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 [26] 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.

2.5. **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 (begin$\langle\rangle$ and end$\langle\rangle$), authenticate of client (auth-client$\langle c \rangle$) or of resource (auth-resource$\langle r \rangle$), and use of resource by client (use$\langle r\, c \rangle$). Using the past time linear temporal logic with calls and returns (ptCaRet) in [31], one would write it as follows:

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

The overlined operators are abstract variants of temporal operators, in the sense that they are defined on traces that collapse terminated procedure calls (erase subtraces bounded by matched begin/end events). For example, "$\overline{\diamondsuit}\,\mathsf{begin}$" 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 [31]. However, until recently [28], JavaMOP could again only monitor it in its non-parametric form, because of its previous 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 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 could not be monitored using the previous JavaMOP system, but it can be monitored with the new one.

2.6. **Success ratio.** Consider now parametric traces with events success$\langle a \rangle$ and 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. We can specify such a property in JavaMOP and RV by making use of monitor variables and event actions [28]. Indeed, one can add two monitor variables, s and t, and then increment t in each event action and increment s only in the event action of the success event. The underlying parametric monitoring algorithm keeps separation between the various s and t monitor variables, one such pair for each distinct action a, guaranteeing that the correct ones will be accessed.

3. Mathematical Background: Partial Functions, Least Upper Bounds (lubs) and lub Closures

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 it turns out that 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.

We recommend the reader who is only interested in our algorithms but not in the details of our subsequent proofs, to read the first two definitions and then jump to Section 4, returning to this section for more mathematical background only when needed.

3.1. **Partial Functions.** This section discusses partial functions and (least) upper bounds over sets of partial functions. The notions and the results discussed in this section are broadly known, and many of their properties are folklore. They can be found in one shape or another in virtually any book on denotational semantics or domain theory. Since we need only a small subset of notions and results on partial functions and (least) upper bounds in this paper, and since we need to fix a uniform notation anyway, we prefer to define and prove everything we need and hereby also make our paper self-contained.

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 $x \in X$).

Definition 1. We let $[X \to V]$ and $[X \rightharpoonup V]$ denote the sets of **total** and of **partial functions** from X to V, respectively. The domain of $\theta \in [X \rightharpoonup V]$ is the set $\mathsf{Dom}(\theta) = \{x \in X \mid \theta(x) \text{ is defined}\}$. Let $\bot \in [X \rightharpoonup V]$ be the map undefined everywhere, that is, $\mathsf{Dom}(\bot) = \emptyset$. If $\theta, \theta' \in [X \rightharpoonup V]$ then:
   (1) θ and θ' are **are compatible** if and only if $\theta(x) = \theta'(x)$ for any $x \in \mathsf{Dom}(\theta) \cap \mathsf{Dom}(\theta')$;
   (2) 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 $\theta'(x) = \theta(x)$;
   (3) θ is **strictly less informative** than θ', written $\theta \sqsubset \theta'$, when $\theta \sqsubseteq \theta'$ and $\theta \neq \theta'$.

The relation of compatibility is reflexive and symmetric, but not transitive. When $\theta, \theta' \in [X \rightharpoonup V]$ are compatible, we let $\theta \sqcup \theta' \in [X \rightharpoonup V]$ denote the partial function whose domain is $\mathsf{Dom}(\theta) \cup \mathsf{Dom}(\theta')$ and which is defined as θ or θ' in each element in its domain. The partial function $\theta \sqcup \theta'$ is called the least upper bound of θ and θ'. We define least upper bounds more generally below. Also, note that $\theta \sqsubseteq \theta'$ and, respectively, $\theta \sqsubset \theta'$, iff θ, θ' compatible and $\mathsf{Dom}(\theta) \subseteq \mathsf{Dom}(\theta')$ and, respectively, $\mathsf{Dom}(\theta) \subset \mathsf{Dom}(\theta')$.

Definition 2. Given $\Theta \subseteq [X \rightharpoonup V]$ and $\theta' \in [X \rightharpoonup V]$,
   (1) 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;
   (2) 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;
   (3) θ' is the **maximum (max)** of Θ iff $\theta' \in \Theta$ and θ' is a lub of Θ.

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

It is known that $([X \rightharpoonup V], \sqsubseteq, \bot)$ is a complete (i.e., any $\sqsubseteq$-chain has a least upper bound) partial order with bottom (i.e., $\bot$).

Definition 3. Given $\Theta \subseteq [X \rightharpoonup V]$, 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 in Sections 3.2 and 3.3 (these sections will present less known results with specific to our particular approach to parametric slicing and monitoring):

Proposition 1. The following hold ($\theta, \theta_1, \theta_2, \theta_3 \in [X \rightharpoonup V]$): $\bot \sqcup \theta$ 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 $\Theta \subseteq [X \rightharpoonup V]$. 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 $\theta_2(x)$ are defined then $\theta_1(x) = \theta_2(x)$;
   (2) If Θ has an upper bound then $\sqcup \Theta$ exists and, for any $x \in X$,

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

Proof. Since Θ has an upper bound $\theta' \in [X \rightharpoonup V]$ iff $\theta \sqsubseteq \theta'$ for any $\theta \in \Theta$, if $\theta_1, \theta_2 \in \Theta$ and $x \in X$ are such that $\theta_1(x)$ and $\theta_2(x)$ are 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 $\theta_2(x)$ are 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 $\theta' \in [X \rightharpoonup V]$ be defined as follows: for any $x \in X$, let

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

First, 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 $\theta_2(x)$ are defined then $\theta_1(x) = \theta_2(x)$. Second, θ' is an upper bound for Θ: indeed, if $\theta \in \Theta$ and $x \in X$ such that $\theta(x)$ is 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 $\theta'(x)$ is defined for some $x \in X$, that is, $\theta(x)$ is defined for some $\theta \in \Theta$ and $\theta'(x) = \theta(x)$, then $\theta''(x)$ is also defined and $\theta'(x) = \theta(x)$ (as $\theta \sqsubseteq \theta''$), so $\theta' \sqsubseteq \theta''$. $\square$

Proposition 3. The following hold:
   (1) The empty set of partial functions $\emptyset \subseteq [X \rightharpoonup V]$ has upper bounds and $\sqcup \emptyset = \bot$;
   (2) The one-element sets have upper bounds and $\sqcup \{\theta\} = \theta$ for any $\theta \in [X \rightharpoonup V]$;
   (3) The bottom "$\bot$" does not influence the least upper bounds: $\sqcup(\{\bot\} \cup \Theta) = \sqcup \Theta$ for any $\Theta \subseteq [X \rightharpoonup V]$;
   (4) If $\Theta, \Theta' \subseteq [X \rightharpoonup V]$ such that $\sqcup \Theta'$ exists and for any $\theta \in \Theta$ there is a $\theta' \in \Theta'$ with $\theta \sqsubseteq \theta'$, then $\sqcup \Theta$ exists and $\sqcup \Theta \sqsubseteq \sqcup \Theta'$; for example, 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 $\Theta_i \subseteq [X \rightharpoonup V]$. 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 $\sqcup \cup \{\Theta_i \mid i \in I\} \sqsubseteq \sqcup\{\sqcup\Theta_i \mid i \in I\}$. $\square$

3.2. **Least Upper Bounds of Families of Sets of Partial Maps.** Motivated by requirements and optimizations of our trace slicing and monitoring algorithms in Sections 6 and 8, in this section and the next we define several less known notions and results. We are actually not aware of other places where these notions are defined, so they could be novel and specific to our approach to parametric trace slicing and monitoring.

We first extend the notion of least upper bound (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 4. If $\{\Theta_i\}_{i \in I}$ is a family of sets in $[X \rightharpoonup V]$, then we let the **least upper bound** (also **lub**) of $\{\Theta_i\}_{i \in I}$ be defined as:

$$\sqcup\{\Theta_i \mid i \in I\} \stackrel{\text{def}}{=} \{\sqcup\{\theta_i \mid i \in I\} \mid \theta_i \in \Theta_i \text{ for each } i \in I \text{ such that } \sqcup\{\theta_i \mid i \in I\} \text{ exists}\}.$$

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 $(M, *_- : M \times M \rightarrow M, 1)$ 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 following facts hold, where $\Theta, \Theta_1, \Theta_2, \Theta_3 \subseteq [X \rightharpoonup V]$:

(1) $\sqcup\emptyset = \{\bot\}$, where, in this case, the empty set $\emptyset \subseteq \mathcal{P}([X \rightharpoonup V])$ is meant;
(2) $\sqcup\{\Theta\} = \Theta$; in particular $\sqcup\{\emptyset\} = \emptyset$ when the empty set $\emptyset \subseteq [X \rightharpoonup V]$ is meant;
(3) $\sqcup\{\{\theta\} \mid \theta \in \Theta\} = \begin{cases} \{\sqcup\Theta\} & \text{if } \Theta \text{ has a lub, and} \\ \emptyset & \text{if } \Theta \text{ does not have a lub;} \end{cases}$
(4) $\emptyset \sqcup \Theta = \emptyset$, where the empty set $\emptyset \subseteq [X \rightharpoonup V]$ is meant;
(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 lubs of sets $\{\theta_i \mid i \in I\}$ containing one element θ_i from each of the 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 $\Theta \subseteq [X \rightharpoonup V]$ 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 $\Theta_1 \sqcup \Theta_3$. $\square$

Proposition 5. *Let $\{\Theta_i\}_{i \in I}$ be a family of sets of partial maps in $[X \rightharpoonup V]$ 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 4 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 4 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:

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

Since $\{I_j\}_{j \in J}$ is a partition of I, the indexes i_j generated by "for each $j \in J$ and $i_j \in I_j$" cover precisely all the indexes $i \in I$. 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 $\sqcup\{\Theta_i \mid i \in I\}$. $\square$

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 $\Theta_1 \sqcup (\Theta_2 \sqcup \Theta_3) = (\Theta_1 \sqcup \Theta_2) \sqcup \Theta_3 = \sqcup\{\Theta_i \mid i \in \{1,2,3\}\}$. $\square$

3.3. **Least Upper Bound Closures.** We next define lub closures of sets of partial maps, a crucial operation for the algorithms discussed next in the paper.

Definition 5. $\Theta \subseteq [X \rightharpoonup V]$ is **lub closed** if and only if $\sqcup\Theta' \in \Theta$ for any $\Theta' \subseteq \Theta$ admitting upper bounds.

Proposition 6. $\{\bot\}$ and $\{\bot, \theta\}$ are lub closed ($\theta \in [X \rightharpoonup V]$).

Proof. It follows easily from Definition 5, 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 $\Theta = \{\theta\}$). $\square$

Proposition 7. If $\Theta \subseteq [X \rightharpoonup V]$ and $\{\Theta_i\}_{i \in I}$ is a family of sets of partial functions in $[X \rightharpoonup V]$, 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) $\cap\{\Theta' \mid \Theta' \subseteq [X \rightharpoonup V]$ 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 5 and using $\sqcup\emptyset = \bot$ (*1.* in Proposition 3).

2. follows taking $\Theta' = \Theta$ in Definition 5: $\sqcup\Theta \in \Theta$, so max Θ exists (and equals $\sqcup\Theta$).

3. Definition 4 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 $\{i\} \cup (I\setminus\{i\})$. By Proposition 5, $\sqcup\{\Theta_i \mid i \in I\} = \Theta_i \sqcup (\sqcup\{\Theta_j \mid j \in I\setminus\{i\}\})$. The proof above also implies that $\sqcup\{\Theta_j \mid j \in I\setminus\{i\}\}$ is lub closed, so by $1.$ we get that $\perp \in \sqcup\{\Theta_j \mid j \in I\setminus\{i\}\}$. Finally, $6.$ in Proposition 4 implies $\Theta_i \sqsubseteq \Theta_i \sqcup (\sqcup\{\Theta_j \mid j \in I\setminus\{i\}\})$, 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 4 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 5), we let $\overline{\Theta}$ denote the set $\cap\{\Theta' \mid \Theta' \subseteq [X \rightharpoonup V]$ 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 $\overline{\Theta}$. $\qquad\square$

Definition 6. Given $\theta' \in [X \rightharpoonup V]$ and $\Theta \subseteq [X \rightharpoonup V]$, let

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

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

Proposition 8. If $\theta, \theta', \theta'', \theta_1, \theta_2 \in [X \rightharpoonup V]$ and if $\Theta \subseteq [X \rightharpoonup V]$ is lub closed, then:

(1) $(\theta']_\Theta$ is lub closed and $\max(\theta']_\Theta$ exists;
(2) If $\theta' \in \{\theta\} \sqcup \Theta$ then $\{\theta'' \mid \theta'' \in \Theta \text{ and } \theta' = \theta \sqcup \theta''\}$ 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 $\{\theta'' \mid \theta'' \in \Theta \text{ and } \theta' = \theta \sqcup \theta''\}$. 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 $\bot$, 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 $\theta_2 = \theta \sqcup \theta_1 = \theta_1$. $\qquad\square$

Definition 7. Given $\Theta \subseteq [X \rightharpoonup V]$, we let $\overline{\Theta}$, the **least upper bound (lub) closure** of Θ, be defined as follows:

$$\overline{\Theta} \stackrel{\text{def}}{=} \cap\{\Theta' \mid \Theta' \subseteq [X \rightharpoonup V] \text{ with } \Theta \subseteq \Theta' \text{ and } \Theta' \text{ is lub closed}\}.$$

Proposition 9. The next hold ($\Theta \subseteq [X \rightharpoonup V], \theta \in [X \rightharpoonup V]$):

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

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

Proposition 10. The lub closure map $\overline{\cdot} : 2^{[X \rightharpoonup V]} \rightarrow 2^{[X \rightharpoonup V]}$ is a closure operator, that is, for any $\Theta, \Theta_1, \Theta_2 \subseteq [X \rightharpoonup V]$,

  (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 $\overline{\Theta_1} \subseteq \overline{\Theta_2}$. $\qquad\square$

Proposition 11. $\overline{\cup\{\Theta_i \mid i \in I\}} = \sqcup\{\overline{\Theta_i} \mid i \in I\}$ for any family $\{\Theta_i\}_{i \in I}$ of partial functions in $[X \rightharpoonup V]$.

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. $\qquad\square$

Corollary 2. For any $\theta \in [X \rightharpoonup V]$ and $\Theta \subseteq [X \rightharpoonup V]$, equality $\overline{\{\theta\} \cup \Theta} = \{\bot, \theta\} \sqcup \overline{\Theta}$ holds.

Proof. $\overline{\{\theta\} \cup \Theta} = \overline{\{\theta\}} \sqcup \overline{\Theta}$ by Proposition 11, and $\overline{\{\theta\}} = \{\bot, \theta\}$ by *3.* in Proposition 9. $\qquad\square$

Corollary 3. If $\Theta \subseteq [X \rightharpoonup V]$ 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, $\overline{\Theta} = \{\bot, \theta_1\} \sqcup \{\bot, \theta_2\} \sqcup \cdots \{\bot, \theta_n\}$; in obtaining that, we used *2.* in Proposition 9 and *1.* in Corollary 1. The result follows now by *6.* in Proposition 7. $\square$

4. 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.

4.1. **The Non-Parametric Case.** We next introduce non-parametric events, traces and properties, which will serve as an infrastructure for their parametric variants in Section 4.2.

Definition 8. 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)* Like in Section 2.1, 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 the set of events $\mathcal{E}$ is $\{\mathsf{acquire}, \mathsf{release}, \mathsf{begin}, \mathsf{end}\}$ and execution traces corresponding to this resource are sequences begin acquire acquire release end begin end, begin acquire acquire, begin acquire release acquire end, etc. For the time being, there are no "good" or "bad" execution traces. $\square$

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. The validating (or matching) 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 (or failing) traces, because

they are not in the language of the regular expression. However, there are two interesting types of such "failing" traces: ones which may still lead to a matching 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 matching 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. Section 2.6 showed an example of a property where the space of trace categories was the set of rational numbers between 0 and 1.

Definition 9. An $\mathcal{E}$-**property** P, or simply a (base or non-parametric) **property**, is a function $P : \mathcal{E}^* \to \mathcal{C}$ partitioning the set of traces into (verdict) categories $\mathcal{C}$.

It is common, though not enforced, that the set of property verdict categories $\mathcal{C}$ in Definition 9 includes validating (or similar), violating (or similar), and don't know (or ?) categories. In general, $\mathcal{C}$ can be any set, finite or infinite.

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 {match, fail, don't know} and, for a given regular expression E, define its associated property $P_E : \mathcal{E}^* \to \mathcal{C}$ as follows: $P_E(w) =$ match iff w is in the language of E, $P_E(w) =$ fail iff there is no $w' \in \mathcal{E}^*$ such that $w\,w'$ is in the language of E, and $P_E(w) =$ don't know otherwise; this is the monitoring semantics of regular expressions in the JavaMOP and RV systems [28, 14, 27].

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

Similarly, one can have various verdict categories for linear temporal logic (LTL). For example, one can report violation when a bad prefix is encountered, can report validation when a good prefix is encountered, and can report inconclusive when neither of the above; this is the current monitoring semantics of monitoring LTL in JavaMOP and RV [28, 27]. Semantical and algorithmic aspects regarding LTL monitoring can be found, e.g., in [21, 32, 33, 7].

In some applications, one may not be interested in certain categories of traces, such as in those classified as don't know, don't care, or inconclusive; if that is the case, then those applications can simply ignore these, like Tracematches, JavaMOP and RV 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. We believe that our definition of property in Definition 9 is general enough to allow us to claim that our results are specification-formalism-independent.

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).

4.2. **The Parametric Case.** Events often carry concrete data instantiating 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 acquire$\langle r \mapsto r_1 \rangle$, 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 begin$\langle \perp \rangle$ and end$\langle \perp \rangle$, where $\perp$, the partial map undefined everywhere, instantiates no parameter.   □

Recall from Definition 1 that $[A \to B]$ and $[A \rightharpoonup B]$ denote the sets of total and, respectively, partial functions from A to B.

Definition 10. (Parametric events and traces). Let X be a set of **parameters** and let V be a set of corresponding **parameter values**. If $\mathcal{E}$ is a set of base events like in Definition 8, then let $\mathcal{E}\langle X \rangle$ denote the set of corresponding **parametric events** $e\langle \theta \rangle$, where e is a base event in $\mathcal{E}$ and θ is a partial function in $[X \rightharpoonup V]$. 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 implicit.

Example. *(part 4)* A parametric trace for our running example can be the following: begin$\langle \perp \rangle$ acquire$\langle \theta_1 \rangle$ acquire$\langle \theta_2 \rangle$ acquire$\langle \theta_1 \rangle$ release$\langle \theta_1 \rangle$ end$\langle \perp \rangle$ begin$\langle \perp \rangle$ acquire$\langle \theta_2 \rangle$ release$\langle \theta_2 \rangle$ end$\langle \perp \rangle$, 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 $\tau\restriction_{r_2}$ instead of $\tau\restriction_{r \mapsto r_2}$, etc. This notation is formally introduced in the next section, as Notation 1. With this notation, the above trace becomes begin$\langle \rangle$ acquire$\langle r_1 \rangle$ acquire$\langle r_2 \rangle$ acquire$\langle r_1 \rangle$ release$\langle r_1 \rangle$ end$\langle \rangle$ begin$\langle \rangle$ acquire$\langle r_2 \rangle$ release$\langle r_2 \rangle$ 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.   □

Recall from Definition 1 that two partial maps of same source and target are compatible when they do not disagree on any of the elements on which they are both defined, and that one is less informative than another, written $\theta \sqsubseteq \theta'$, when they are compatible and the domain of the former in included in the domain of the latter. With the notation in the example above we have, for our running example, that $\langle \rangle$ is compatible with $\langle r_1 \rangle$ and with $\langle r_2 \rangle$, but $\langle r_1 \rangle$ and $\langle r_2 \rangle$ are not compatible; moreover, $\langle \rangle \sqsubseteq \langle r_1 \rangle$ and $\langle \rangle \sqsubseteq \langle r_2 \rangle$.

Definition 11. (Trace slicing) Given parametric trace $\tau \in \mathcal{E}\langle X \rangle^*$ and partial function θ in $[X \rightarrow V]$, we let the θ**-trace slice** $\tau\!\restriction_\theta \in \mathcal{E}^*$ be the non-parametric trace in $\mathcal{E}^*$ defined as:

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

Therefore, the trace slice $\tau\!\restriction_\theta$ 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. It is crucial to discard parameter instances that are not relevant to θ during the slicing, including those more informative than θ, in order to achieve a correct slice for θ: in our running example, the trace slice for $\langle\rangle$ should contain only begin and end events and no acquire or release. Otherwise, the acquire and release of different resources will interfere with each other in the trace slice for $\langle\rangle$.

One should not confuse extracting/abstracting traces from executions with slicing traces. The former determines the events to include in the trace, as well as parameter instances carried by events, while the latter dispatches each event in the given trace to corresponding trace slices according to the event's parameter instance. Different abstractions may result in different parametric traces from the same execution and thus may lead to different trace slices for the same parameter instance θ. For the (map, collection, iterator) example in Section 1, $X = \{m, c, i\}$ and an execution may generate the following parametric trace: createColl$\langle m_1, c_1 \rangle$ createIter$\langle c_1, i_1 \rangle$ next$\langle i_1 \rangle$ updateMap$\langle m_1 \rangle$. The trace slice for $\langle m_1 \rangle$ is updateMap for this parametric trace. Now suppose that we are only interested in operations on maps. Then $X = \{m\}$ and the trace abstracted from the execution generating the above trace is createColl$\langle m_1 \rangle$ updateMap$\langle m_1 \rangle$, in which events and parameter bindings irrelevant to m are removed. Then the trace slice for $\langle m_1 \rangle$ is createColl updateMap. In this paper we focus only on trace slicing; more discussion about trace abstraction can be found in [15].

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 sort of a universally quantified property over base events, but, unfortunately, the underlying specification formalism may not allow such 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 data quantification by default. We say "by default" because, in some cases, there are attempts to add data quantification; for example, [23] investigates the implications and the necessary restrictions resulting from adding quantifiers to LTL, and [33] investigates a finite-trace variant of parametric LTL together with a translation to parametric alternating automata.

Definition 12. Let X be a set of parameters together with their corresponding parameter values V, like in Definition 10, and let $P : \mathcal{E}^* \to \mathcal{C}$ be a non-parametric property like in Definition 9. Then we define the **parametric property** $\Lambda X . P$ as the property (over traces $\mathcal{E}\langle X \rangle^*$ and categories $[[X \rightarrow V] \to \mathcal{C}]$)

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

defined as

$$(\Lambda X . P)(\tau)(\theta) = P(\tau\!\restriction_\theta)$$

for any $\tau \in \mathcal{E}\langle X \rangle^*$ and any $\theta \in [X \rightharpoonup V]$. If $X = \{x_1, ..., x_n\}$ we may write $\Lambda x_1, ..., x_n \,.\, P$ instead of $(\Lambda\{x_1, ..., x_n\} \,.\, P$. Also, if P_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 $P : \mathcal{E}^* \rightarrow \mathcal{C}$ are therefore properties taking traces in $\mathcal{E}\langle X \rangle^*$ to categories $[[X \rightharpoonup V] \rightarrow \mathcal{C}]$, i.e., to function domains from parameter instances to base property categories. $\Lambda X \,.\, P$ is defined as if many instances of P are observed at the same time on the parametric trace, one property instance for each parameter instance, each property instance concerned with its events only, dropping the unrelated ones.

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 9 – i.e., the JavaMOP/RV semantic approach to parametric monitoring [14]). Since we want P to hold for any resource instance, we define the following parametric property:

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

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 validating for parameter instance θ_1 and violating for parameter instance θ_2. $\qquad\square$

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 "$a_2 b_1 c_1$". The function $\bot$ then corresponds to the empty sequence.

Example. Here is a parametric trace with events parametric in $\{a, b, c\}$, where the parameters take values in the set $\{a_1, a_2, b_1, c_1\}$:

$$\tau = e_1\langle a_1 \rangle\ e_2\langle a_2 \rangle\ e_3\langle b_1 \rangle\ e_4\langle a_2 b_1 \rangle\ e_5\langle a_1 \rangle\ e_6\langle\rangle\ e_7\langle b_1 \rangle\ e_8\langle c_1 \rangle\ e_9\langle a_2 c_1 \rangle\ e_{10}\langle a_1 b_1 c_1 \rangle\ e_{11}\langle\rangle.$$

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 [25], Tracematches [1], JavaMOP [14] and RV [27]) 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, i.e., if $e\langle\theta\rangle$ and $e\langle\theta'\rangle$ appear in a

parametric trace, then $\mathsf{Dom}(\theta) = \mathsf{Dom}(\theta')$. While this restriction is reasonable and sometimes useful, our trace slicing and monitoring algorithms in this paper do not need it.   □

Recall from Definition 11 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:

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

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

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, $\Theta_\tau = \overline{\{\theta \mid \theta \in [X \rightharpoonup V],\ e\langle\theta\rangle \in \tau\}}$.

Proposition 12. For any parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$, the set Θ_τ is a finite lub closed set.

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:

$$
\begin{aligned}
\Theta_{\tau\, e\langle\theta\rangle} &= \overline{\{\theta' \mid \theta' \in [X \rightharpoonup V],\ e'\langle\theta'\rangle \in \tau\, e\langle\theta\rangle\}}\\
&= \overline{\{\theta\} \cup \{\theta' \mid \theta' \in [X \rightharpoonup V],\ e'\langle\theta'\rangle \in \tau\}}\\
&= \overline{\{\theta\} \cup \Theta_\tau}\\
&= \{\bot, \theta\} \sqcup \overline{\Theta_\tau}\\
&= \{\bot, \theta\} \sqcup \Theta_\tau.
\end{aligned}
$$

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 $\theta \in [X \rightharpoonup V]$, the equality $\tau\!\restriction_\theta = \tau\!\restriction_{\max(\theta]_{\Theta_\tau}}$ holds.

Proof. We prove the following more general result:

"Let $\Theta \subseteq [X \rightharpoonup V]$ be lub closed and let $\theta \in [X \rightharpoonup V]$;
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 $\theta \in [X \rightharpoonup V]$, we pick Θ to be Θ_τ. We prove the general result by induction on the *length* of τ:

Algorithm $\mathbb{A}\langle X \rangle$
Input: parametric trace $\tau \in \mathcal{E}\langle X \rangle^*$
Output: map $\mathbb{T} \in [[X \to V] \to \mathcal{E}^*]$ and set $\Theta \subseteq [X \to V]$

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

Figure 2: Parametric trace slicing algorithm $\mathbb{A}\langle X \rangle$.

- 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. $\qquad\square$

6. Algorithm for Online Parametric Trace Slicing

Definition 11 illustrates a way to slice a parametric trace for *given* parameter bindings. However, it is not suitable for online trace slicing, where the trace is observed incrementally and no future knowledge is available, because we cannot know all possible parameter instances θ apriori. 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 $\mathbb{T} \in [[X \to V] \to \mathcal{E}^*]$ of finite domain that serves as a quick lookup table for all slices of τ. More precisely, Theorem 1 shows that, for any $\theta \in [X \to V]$, 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 (see Definition 13 for Θ_τ). 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 $\theta \in [X \to V]$ 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 (Definition 11) would do.

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

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

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).

Figure 2 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 $\theta' \in [X \rightharpoonup V]$ 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 $\mathbb{T}$ that is less informative or as informative as θ'; the Θ data-structure is then extended in line 6 (see Proposition 13 for why this way).

Example. Consider again the sample trace in Section 5 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 1 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 Θ stays lub closed. $\square$

$\mathbb{A}\langle X\rangle$ computes trace slices for all combinations of parameter instances observed in parametric trace events. Its complexity is therefore $O(n \times m)$ where n is the length of the trace and m is the number of all possible parameter combinations. However, $\mathbb{A}\langle X\rangle$ is not intended to be implemented directly; it is only used as a correctness backbone for other trace analysis algorithms, such as the monitoring algorithms discussed below. An alternative and apparently more efficient solution is to only record trace slices for parameter instances that actually appear in the trace (instead of for all combinations of them), and then construct the slice for a given parameter instance by combining such trace slices for compatible parameter instances. However, the complexity of constructing all possible trace slices at the end using such an algorithm is also $O(n \times m)$, so it would not bring any benefit overall compared to $\mathbb{A}\langle X\rangle$. In addition, $\mathbb{A}\langle X\rangle$ is more suitable as a backbone for developing online monitoring algorithms such as those in Section 7, because each event is sent to its slices (that will be consumed by corresponding monitors) and never touched again.

$\mathbb{A}\langle X\rangle$ compactly and uniformly captures several special cases and subcases that are worth discussing. The discussion below can be formalized as an inductive (on the length of 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 $\Theta \subseteq \{\perp, \theta\} \sqcup \Theta$. 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 $\mathsf{Dom}(\mathbb{T}) = \Theta = \Theta_\epsilon = \{\perp\}$ initially and since 5. and 7. in Proposition 4 imply $\Theta \cup (\{\theta\} \sqcup \Theta) = \{\perp, \theta\} \sqcup \Theta$ while Proposition 13 states that $\Theta_{\tau\, e\langle\theta\rangle} = \{\perp, \theta\} \sqcup \Theta_\tau$, 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 $\perp \in \Theta$. Thanks to Proposition 14, 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 wrongly 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 $\mathbb{A}\langle X\rangle$.

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) maintained by the algorithm $\mathbb{A}\langle X\rangle$ in Figure 2 after it processes τ.

Theorem 1. With the notation in Definition 14, 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) $\tau\!\!\upharpoonright_\theta = \mathbb{A}\langle X\rangle(\tau).\mathbb{T}(\max{(\theta]_{\mathbb{A}\langle X\rangle(\tau).\Theta}})$ for any $\theta \in [X \rightharpoonup V]$.

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.* 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.*:

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

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 duplicating 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,

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

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 $(\tau\,e\langle\theta\rangle)\upharpoonright_{\theta'} = \tau\upharpoonright_{\theta'}$; furthermore,

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

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:

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

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

7. Monitors and 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 (deterministic) Moore machine [29] that allows infinitely many states:

Definition 15. A **monitor** M is a tuple $(S, \mathcal{E}, \mathcal{C}, \iota, \sigma : S \times \mathcal{E} \to S, \gamma : S \to \mathcal{C})$, where S is a set of states, $\mathcal{E}$ is a set of input events, $\mathcal{C}$ is a set of output categories, $\iota \in S$ 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 which performs an NFA-to-DFA construction on the fly, as events are received. Such a monitor generates only those states in the DFA that are needed by the monitored execution trace. Moreover, the monitor only needs to store one such state of the DFA, i.e., set of states in the NFA, namely the current one: once an event is received, the next state is (deterministically) computed and the old one is discarded. Therefore, assuming that one needs constant space to store a state of the original NFA, then the memory needed by this monitor is linear in the number of states of the NFA. An alternative and probably more conventional monitor could be one which generates the corresponding DFA statically, paying upfront the exponential price in both time and space. As empirically suggested by [35], if one is able to statically generate and store the corresponding DFA then one should most likely take this route, because in practice it tends to be much faster to jump to a known next state than to compute it.

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 [26] 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. Nevertheless, what is common to all monitors is that they can classify traces into categories. When a monitor does not have enough information about a trace to put it in a category of interest, we can assume that it actually categorizes it as a "don't know" trace, where "don't know" can be regarded as a special category; this is similar to regarding partial functions as total functions by adding a special "undefined" value in their codomain. The following is therefore natural:

Definition 16. Monitor $M = (S, \mathcal{E}, \mathcal{C}, \iota, \sigma, \gamma)$ is a **monitor for property** $P : \mathcal{E}^* \to \mathcal{C}$ if and only if $\gamma(\sigma(\iota, w)) = P(w)$ for each $w \in \mathcal{E}^*$.

A property can be associated to each monitor, in a similar style to which we can associate a language to each automaton:

Definition 17. Monitor $M = (S, \mathcal{E}, \mathcal{C}, \iota, \sigma, \gamma)$ defines **the M-property** $\mathcal{P}_M : \mathcal{E}^* \to \mathcal{C}$ as follows: $\mathcal{P}_M(w) = \gamma(\sigma(\iota, w))$ for each $w \in \mathcal{E}^*$.

The following result is straightforward, it follows immediately from Definitions 16 and 17. The only reason we frame it as a numbered proposition is because we need to refer to it in the proof of Corollary 5.

Proposition 15. With the notation in Definition 17, monitor M is indeed a monitor for its corresponding M-property $\mathcal{P}_M$. Moreover, a monitor can only be a monitor for one property, that is, if M is a monitor for property P then $P = \mathcal{P}_M$.

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

Definition 18. Property $P : \mathcal{E}^* \to \mathcal{C}$ defines **the P-monitor** $\mathcal{M}_P = (S_P, \mathcal{E}, \mathcal{C}, \iota_P, \sigma_P, \gamma_P)$ as follows:

$S_P = \mathcal{E}^*$,
$\iota_P = \epsilon$,
$\sigma_P(w, e) = we$ for each $w \in S_P = \mathcal{E}^*$ and $e \in \mathcal{E}$,
$\gamma_P(w) = P(w)$ for each $w \in S_P = \mathcal{E}^*$.

Thus, $\mathcal{M}_P$ holds traces as states, appends events to them as transition and, as output, it looks up the category of the corresponding trace using P. The following results are also straightforward and, again, we frame them as numbered propositions only because we will refer to them later.

Proposition 16. With the notation in Definition 18, the monitor $\mathcal{M}_P$ is indeed a monitor for property P.

Proof. It follows from the sequence of equalities $\gamma_P(\sigma_P(\iota_P, w)) = \gamma_P(\sigma_P(\epsilon, w)) = \gamma_P(\epsilon w) = \gamma_P(w) = P(w)$. $\square$

Proposition 17. With the notations in Definitions 17 and 18, $\mathcal{P}_{\mathcal{M}_P} = P$ for any property $P : \mathcal{E}^* \to \mathcal{C}$.

Proof. $\mathcal{P}_{\mathcal{M}_P}(w) = \gamma_P(\sigma_P(\iota_P, w)) = P(w)$ for any $w \in \mathcal{E}^*$. $\square$

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, though.

Definition 19. Monitors M and M' are **property equivalent**, or just **equivalent**, written $M \equiv M'$, iff they are monitors for the same property (see Definition 16). With the notation in Definition 17, we have that $M \equiv M'$ iff $\mathcal{P}_M = \mathcal{P}_{M'}$.

Proposition 18. With the notations in Definitions 17 and 18, $\mathcal{M}_{\mathcal{P}_M} \equiv M$ for any monitor $M = (S, \mathcal{E}, \mathcal{C}, \iota, \sigma, \gamma)$.

Proof. By Definition 19, $\mathcal{M}_{\mathcal{P}_M} \equiv M$ iff $\mathcal{P}_{\mathcal{M}_{\mathcal{P}_M}} = \mathcal{P}_M$, and the latter follows by Proposition 16 taking P to be $\mathcal{P}_M$. $\square$

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 20. Given parameter set X with corresponding values V and a monitor $M = (S, \mathcal{E}, \mathcal{C}, \iota, \sigma : S \times \mathcal{E} \to S, \gamma : S \to \mathcal{C})$, we define the **parametric monitor** $\Lambda X . M$ as the monitor

$$([[X \rightharpoonup V] \rightarrow S], \mathcal{E}\langle X \rangle, [[X \rightharpoonup V] \rightarrow \mathcal{C}], \lambda\theta.\iota, \Lambda X . \sigma, \Lambda X . \gamma),$$

with

$$\Lambda X . \sigma : [[X \rightharpoonup V] \rightarrow S] \times \mathcal{E}\langle X \rangle \to [[X \rightharpoonup V] \rightarrow S]$$
$$\Lambda X . \gamma : [[X \rightharpoonup V] \rightarrow S] \to [[X \rightharpoonup V] \rightarrow \mathcal{C}]$$

defined as

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

$$(\Lambda X \,.\, \gamma)(\delta)(\theta) = \gamma(\delta(\theta))$$

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

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 19. If M is a monitor for P then parametric monitor $\Lambda X \,.\, M$ is a monitor for parametric property $\Lambda X \,.\, P$, or, with the notation in Definition 17, $\mathcal{P}_{\Lambda X \,.\, M} = \Lambda X \,.\, \mathcal{P}_M$.

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

$$\begin{aligned} (\Lambda X \,.\, \sigma)(\lambda\theta.\iota, \tau\, e\langle\theta'\rangle)(\theta) &= (\Lambda X \,.\, \sigma)((\Lambda X \,.\, \sigma)(\lambda\theta.\iota, \tau), e\langle\theta'\rangle)(\theta) \\ &= (\Lambda X \,.\, \sigma)(\sigma(\iota, \tau\restriction_\theta), e\langle\theta'\rangle)(\theta) \\ &= \left\{ \begin{array}{ll} \sigma(\sigma(\iota, \tau\restriction_\theta), e) & \text{if } \theta' \sqsubseteq \theta \\ \sigma(\iota, \tau\restriction_\theta) & \text{if } \theta' \not\sqsubseteq \theta \end{array} \right. \\ &= \left\{ \begin{array}{ll} \sigma(\iota, \tau\restriction_\theta\, e) & \text{if } \theta' \sqsubseteq \theta \\ \sigma(\iota, \tau\restriction_\theta) & \text{if } \theta' \not\sqsubseteq \theta \end{array} \right. \\ &= \sigma(\iota, (\tau\, e\langle\theta'\rangle)\restriction_\theta) \end{aligned}$$

The first equality above follows by the second part of Definition 15), the second by the induction hypothesis, the third by Definition 20, the fourth again by the second part of Definition 15, and the fifth by Definition 11. This concludes our proof. $\qquad\square$

8. Algorithms for Parametric Trace Monitoring

We next propose two monitoring algorithms for parametric properties. Our unoptimized but easier to understand algorithm is easily derived from the parametric trace slicing algorithm in Figure 2. Our second algorithm is an online optimization of the first, which significantly reduces the size of the search space for compatible parameter instances when a new event is received.

8.1. **Unoptimized but Simpler Algorithm.** Analyzing the definition of a parametric monitor (Definition 20), 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 2, we encode functions $[[X \rightharpoonup V] \rightarrow S]$ as tables with entries indexed by parameter instances in $[X \rightharpoonup V]$ and with contents states in S. Following similar arguments

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

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

Figure 3: Parametric monitoring algorithm $\mathbb{B}\langle X\rangle$

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 3 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, 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 2, 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 21. 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 3 after processing τ. Let $\bot \mapsto \iota = \mathbb{B}\langle X\rangle(M)(\epsilon).\Delta \in [[X \rightharpoonup V] \rightharpoonup S]$ be the partial map taking $\bot \in [X \rightharpoonup V]$ to ι and undefined elsewhere.

Corollary 4. The following hold for any $\tau \in \mathcal{E}\langle X\rangle^*$:
  (1) $\mathsf{Dom}(\mathbb{B}\langle X\rangle(M)(\tau).\Delta) = \mathbb{B}\langle X\rangle(M)(\tau).\Theta = \Theta_\tau$;
  (2) $\mathbb{B}\langle X\rangle(M)(\tau).\Delta(\theta) = \sigma(\iota, \tau{\restriction}_\theta)$ and
      $\mathbb{B}\langle X\rangle(M)(\tau).\Gamma(\theta) = \gamma(\sigma(\iota, \tau{\restriction}_\theta))$ for any $\theta \in \mathbb{B}\langle X\rangle(M)(\tau).\Theta$;
  (3) $\sigma(\iota, \tau{\restriction}_\theta) = \mathbb{B}\langle X\rangle(M)(\tau).\Delta(\max{(\theta]_{\mathbb{B}\langle X\rangle(M)(\tau).\Theta}})$ and
      $\gamma(\sigma(\iota, \tau{\restriction}_\theta)) = \mathbb{B}\langle X\rangle(M)(\tau).\Gamma(\max{(\theta]_{\mathbb{B}\langle X\rangle(M)(\tau).\Theta}})$ for any $\theta \in [X \rightharpoonup V]$.

Proof. Follows from Theorem 1 and the discussion above. $\square$

We next show how to associate a formal monitor to the algorithm $\mathbb{B}\langle X\rangle(M)$ in Figure 3:

Definition 22. For the algorithm $\mathbb{B}\langle X\rangle(M)$ in Figure 3, let

$$\mathcal{M}_{\mathbb{B}\langle X\rangle(M)} = (R, \mathcal{E}\langle X\rangle, [[X \rightarrow V] \rightarrow \mathcal{C}], \perp \mapsto \iota, next, out)$$

be the monitor defined as follows:

- $R \subseteq [[X \rightarrow V] \rightarrow S]$ 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
- $next : R \times \mathcal{E}\langle X\rangle \rightarrow R$ and $out : R \rightarrow [[X \rightarrow V] \rightarrow \mathcal{C}]$ are functions defined as follows, where $\tau \in \mathcal{E}\langle X\rangle^*$, $e \in \mathcal{E}$, and $\theta \in [X \rightarrow V]$:

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

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 $\tau \in \mathcal{E}\langle X\rangle^*$, $out(next(\perp \mapsto \iota, \tau))$ and $(\Lambda X\,.\,\gamma)((\Lambda X\,.\,\sigma)(\lambda\theta.\iota, \tau))$ are equal as total functions in $[[X \rightarrow V] \rightarrow \mathcal{C}]$. Let $\theta \in [X \rightarrow V]$; then:

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

The first equality above follows inductively by the definition of *next* (Definition 22), noticing that $\perp \mapsto \iota = \mathbb{B}\langle X\rangle(M)(\epsilon).\Delta$. The second equality follows by the definition of *out* (Definition 22) and the third by *3.* in Corollary 4. The fourth equality above follows inductively by the definition of $\Lambda X\,.\,\sigma$ (Definition 20) and has already been proved as part of the proof of Proposition 19. Finally, the fifth equality follows by the definition of $\Lambda X\,.\,\gamma$ (Definition 20).

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

Corollary 5. 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 Definition 17, Theorem 2 implies that $\mathcal{P}_{\mathcal{M}_{\mathbb{B}\langle X\rangle(M)}} = \mathcal{P}_{\Lambda X\,.\,M}$. By Proposition 19 we have that $\mathcal{P}_{\Lambda X\,.\,M} = \Lambda X\,.\,\mathcal{P}_M$. Finally, since $P = \mathcal{P}_M$ by Proposition 15, we conclude that $\mathcal{P}_{\mathcal{M}_{\mathbb{B}\langle X\rangle(M)}} = \Lambda X\,.\,P$. $\square$

8.2. **Optimized Algorithm.** Algorithm $\mathbb{C}\langle X\rangle$ in Figure 4 refines Algorithm $\mathbb{B}\langle X\rangle$ in Figure 3 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 3). The direct use of $\mathbb{B}\langle X\rangle$ would yield prohibitive runtime overhead when monitoring large traces, because its inner loop requires search for all parameter instances in 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 further optimizations. 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 in order to terminate. Note that in practice $\mathsf{Dom}(\theta)$ is always finite (because the program state is finite).

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

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(θ, θ_{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 θ 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(θ, θ')
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 4: Online parametric monitoring algorithm $\mathbb{C}\langle X \rangle$

$\mathbb{C}\langle X \rangle$ uses three tables: Δ, $\mathcal{U}$ 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 3). 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., $\mathcal{U}(\theta) = \{\theta' \mid \theta' \in \mathsf{Dom}(\Delta) \text{ and } \theta \sqsubset \theta'\}$. Another major difference between $\mathbb{B}\langle X \rangle$ and $\mathbb{C}\langle X \rangle$ is that $\mathbb{C}\langle X \rangle$ does *not* maintain Θ during computation; instead, Θ is implicitly captured by the domain of Δ in $\mathbb{C}\langle X \rangle$. Intuitively, the Θ at the beginning/end of the body of the outer loop in $\mathbb{B}\langle X \rangle$ is the $\mathsf{Dom}(\Delta)$ 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, $\{\theta\} \sqcup \Theta = \{\theta' \mid \theta' \in \Theta \text{ and } \theta \sqsubseteq \theta'\} \subseteq \Theta$, so the lines 3 to 6 in $\mathbb{B}\langle X \rangle$ become precisely the lines 16 to 19 in $\mathbb{C}\langle X \rangle$. In the other case, when 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 14). 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 $\mathcal{U}$ (lines 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 3: 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 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, if $\mathsf{Dom}(\theta)$ is finite 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 $\theta \in [X \rightharpoonup V]$ and thus the loop at line 9 in main also terminates. Assuming that running the base monitor M takes constant time, the worst case complexity of $\mathbb{C}\langle X \rangle(M)$ is $O(k \times l)$ to process $e\langle\theta\rangle$, where k is $2^{|\mathsf{Dom}(\theta)|}$ and l is the number of incompatible parameter instances in τ. Parametric properties often have a fixed and small number of parameters, in which case k is not significant. Depending on the trace, l can unavoidably grow arbitrarily large; in the worst case, each event may carry an instance incompatible with the previous ones.

Lemma 1. In the algorithm $\mathbb{C}\langle X \rangle$ in Figure 4, $\mathcal{U}(\theta) = \{\theta' \mid \theta' \in \mathsf{Dom}(\Delta) \text{ and } \theta \sqsubset \theta'\}$ before and after each execution of defineTo, for all $\theta \in [X \rightharpoonup V]$.

Proof. By how $\mathbb{C}\langle X \rangle$ is initialized, for any $\theta \in [X \rightharpoonup V]$ we have $\emptyset = \mathcal{U}(\theta) = \{\theta' \mid \theta' \in \mathsf{Dom}(\Delta) \text{ and } \theta \sqsubset \theta'\}$ before the first execution of defineTo. Now suppose that $\mathcal{U}(\theta) = \{\theta' \mid \theta' \in \mathsf{Dom}(\Delta) \text{ and } \theta \sqsubset \theta'\}$ for any $\theta \in [X \rightharpoonup V]$ 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 $\theta'' \in [X \rightharpoonup V]$ with $\theta'' \sqsubset \theta$, we still have $\mathcal{U}(\theta) = \{\theta' \mid \theta' \in \mathsf{Dom}(\Delta) \text{ and } \theta \sqsubset \theta'\}$ for any $\theta \in [X \rightharpoonup V]$ after the execution of defineTo. Also, the only way $\mathbb{C}\langle X \rangle$ can add a new parameter instance θ into $\mathsf{Dom}(\Delta)$ is by using defineTo. Therefore the lemma holds. $\square$

The next theorem proves the correctness of $\mathbb{C}\langle X \rangle$. Before we state and prove it, let us recall some previously introduced notation and also introduce some new useful notation. First, recall from Definition 21 that $\mathbb{B}\langle X \rangle(M)(\tau).\Delta$ and $\mathbb{B}\langle X \rangle(M)(\tau).\Gamma$ are the Δ and Γ

data-structures of $\mathbb{B}\langle X\rangle(M)$ after it processes trace τ. Also, recall that we fixed parametric trace τ and event $e\langle\theta\rangle$. For clarity, let $\mathcal{U}_\mathbb{C}$, $\Delta_\mathbb{C}$, and $\Gamma_\mathbb{C}$ be the three data-structures maintained by $\mathbb{C}\langle X\rangle(M)$ (in other words, we index the data-structures with the symbol $\mathbb{C}$). Let $\Delta_\mathbb{C}^b$ and $\Gamma_\mathbb{C}^b$ be the $\Delta_\mathbb{C}$ and $\Gamma_\mathbb{C}$ when $\mathsf{main}(e\langle\theta\rangle)$ begins ("b" stays for "at the beginning"); let $\Delta_\mathbb{C}^e$ and $\Gamma_\mathbb{C}^e$ be the $\Delta_\mathbb{C}$ and $\Gamma_\mathbb{C}$ when $\mathsf{main}(e\langle\theta\rangle)$ ends ("e" stays for "at the end"; and let $\Delta_\mathbb{C}^m$ and $\mathcal{U}_\mathbb{C}^m$ be the $\Delta_\mathbb{C}$ and $\mathcal{U}_\mathbb{C}$ when $\mathsf{main}(e\langle\theta\rangle)$ reaches line 16 ("m stays for "in the middle").

Theorem 3. The following hold:

(1) $\mathsf{Dom}(\Delta_\mathbb{C}^m) = \{\bot, \theta\} \sqcup \mathsf{Dom}(\Delta_\mathbb{C}^b)$;

(2) $\Delta_\mathbb{C}^m(\theta') = \Delta_\mathbb{C}^m(\max(\theta']_{\mathsf{Dom}(\Delta_\mathbb{C}^b)})$, for all $\theta' \in \mathsf{Dom}(\Delta_\mathbb{C}^m)$;

(3) If $\Delta_\mathbb{C}^b = \mathbb{B}\langle X\rangle(M)(\tau).\Delta$ and $\Gamma_\mathbb{C}^b = \mathbb{B}\langle X\rangle(M)(\tau).\Gamma$, then $\Delta_\mathbb{C}^e = \mathbb{B}\langle X\rangle(M)(\tau\, e\langle\theta\rangle).\Delta$ and $\Gamma_\mathbb{C}^b = \mathbb{B}\langle X\rangle(M)(\tau\, e\langle\theta\rangle).\Gamma$.

Proof. Let $\Theta_\mathbb{C} = \mathsf{Dom}(\Delta_\mathbb{C}^b) = \mathsf{Dom}(\Delta_\mathbb{B}(\tau))$ 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 whether θ is in $\Theta_\mathbb{C}$ or not. If $\theta \in \Theta_\mathbb{C}$ then the lines 2 to 14 are skipped and $\mathsf{Dom}(\Delta_\mathbb{C})$ remains unchanged, that is, $\{\bot, \theta\} \sqcup \Theta_\mathbb{C} = \Theta_\mathbb{C} = \mathsf{Dom}(\Delta_\mathbb{C}^b) = \mathsf{Dom}(\Delta_\mathbb{C}^m)$ when $\mathsf{main}(e\langle\theta\rangle)$ 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, $\mathsf{Dom}(\Delta_\mathbb{C}^m)$ 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 $\Delta_\mathbb{C}^m$ remains defined on $\Theta_\mathbb{C}$, we conclude that $\Delta_\mathbb{C}^m$ 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 $\theta' \in \mathsf{Dom}(\Delta_\mathbb{C}^m)$, so the result follows. Suppose now that $\theta \notin \Theta_\mathbb{C}$. By *1.* and its proof, each $\theta' \in \mathsf{Dom}(\Delta_\mathbb{C}^m)$ 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 $\Delta_{\mathbb{C}}^e = \Delta_{\mathbb{B}}(\tau e)$. 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) $\mathsf{Dom}(\Delta_{\mathbb{C}}^e) = \{\bot, \theta\} \sqcup \Theta_{\mathbb{C}}$;
2) $\forall\, \theta' \in \{\theta\} \sqcup \Theta_{\mathbb{C}},\ \Delta_{\mathbb{C}}^e(\theta') = \sigma(\Delta_{\mathbb{C}}^b(\max, (\theta']_{\Theta_{\mathbb{C}}}), e)$;
3) $\forall\, \theta' \in \Theta_{\mathbb{C}} - \{\theta\} \sqcup \Theta_{\mathbb{C}},\ \Delta_{\mathbb{C}}^e(\theta') = \Delta_{\mathbb{C}}^b(\theta')$.
By *1.*, we have $\mathsf{Dom}(\Delta_{\mathbb{C}}^m) = \{\bot, \theta\} \sqcup \Theta_{\mathbb{C}}$. Since lines 16 to 19 do not change $\mathsf{Dom}(\Delta_{\mathbb{C}})$, $\mathsf{Dom}(\Delta_{\mathbb{C}}^e) = \mathsf{Dom}(\Delta_{\mathbb{C}}^m) = \{\bot, \theta\} \sqcup \Theta_{\mathbb{C}}$. 1) holds.

By *2.* and Lemma 1, $\Delta_{\mathbb{C}}^m(\theta') = \Delta_{\mathbb{C}}^b(\max, (\theta']_{\Theta_{\mathbb{C}}})$ for any $\theta' \in \mathsf{Dom}(\Delta_{\mathbb{C}}^m)$. Also, notice that line 17 sets $\Delta_{\mathbb{C}}(\theta')$ to $\sigma(\Delta_{\mathbb{C}}(\theta'), e)$, which is $\sigma(\Delta_{\mathbb{C}}^b(\max, (\theta']_{\Theta_{\mathbb{C}}}), e)$, 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 $\{\theta\} \cup \mathcal{U}_{\mathbb{C}}^m(\theta) = \{\theta\} \sqcup \Theta_{\mathbb{C}}$. Since $\mathsf{Dom}(\Delta_{\mathbb{C}}^m) = \{\bot, \theta\} \sqcup \Theta_{\mathbb{C}}$, we have $\{\theta\} \sqcup \mathsf{Dom}(\Delta_{\mathbb{C}}^m) = \{\theta\} \sqcup (\{\bot, \theta\} \sqcup \Theta_{\mathbb{C}}) = \{\theta\} \sqcup (\{\theta\} \sqcup \Theta_{\mathbb{C}}) \cup \Theta_{\mathbb{C}})$. By Proposition 4, $\{\theta\} \sqcup \mathsf{Dom}(\Delta_{\mathbb{C}}^m) = (\{\theta\} \sqcup (\{\theta\} \sqcup \Theta_{\mathbb{C}})) \cup (\{\theta\} \sqcup \Theta_{\mathbb{C}}) = (\{\theta\} \sqcup \Theta_{\mathbb{C}}) \cup (\{\theta\} \sqcup \Theta_{\mathbb{C}}) = \{\theta\} \sqcup \Theta_{\mathbb{C}}$. Also, as $\theta \in \mathsf{Dom}(\Delta_{\mathbb{C}}^m)$, we have $\{\theta\} \sqcup \mathsf{Dom}(\Delta_{\mathbb{C}}^m) = \{\theta' \mid \theta' \in \mathsf{Dom}(\Delta_{\mathbb{C}}^m) \text{ and } \theta \sqsubseteq \theta'\} = \{\theta\} \sqcup \mathcal{U}_{\mathbb{C}}^m(\theta)$ by Lemma 1. So $\{\theta\} \cup \mathcal{U}_{\mathbb{C}}^m(\theta) = \{\theta\} \sqcup \Theta_{\mathbb{C}}$. $\qquad\square$

We conclude this section with a discussion on the complexity of the parametric monitoring algorithms $\mathbb{A}\langle X\rangle$ and $\mathbb{C}\langle X\rangle$ above. Note that, in the worst case, to process a newly received parametric event $e\langle\theta\rangle$ after $\mathbb{A}\langle X\rangle$ or $\mathbb{C}\langle X\rangle$ has already processed a parametric trace τ, each of $\mathbb{A}\langle X\rangle$ or $\mathbb{C}\langle X\rangle$ takes at least linear time/space in the number of θ-compatible parameter instances occurring in events in τ. Indeed, $\mathbb{A}\langle X\rangle$ iterates explicitly through all such parameter instances (line 3 in Figure 3), while $\mathbb{C}\langle X\rangle$ optimizes this traversal by only enumerating through maximal parameter instances; in the worst case, we can assume that τ is such that each event comes with a new parameter instance which is maximal, so in the worst case $\mathbb{A}\langle X\rangle$ and $\mathbb{C}\langle X\rangle$ can take linear time/space in τ to process $e\langle\theta\rangle$, which is, nevertheless, bad. Indeed, it means that monitoring some traces is incrementally slower (with no upper bound) as events are received, until the monitor eventually runs out of resources.

Unfortunately, there is nothing to be fundamentally done to avoid the problem above. It is an inherent problem of parametric monitoring. Consider, for example, the "authenticate before use" parametric property specified in Section 2.2 using parametric LTL as $\Lambda k\,.\,\Box(\mathsf{use}\langle k\rangle \to \Diamond\mathsf{authenticate}\langle k\rangle)$; to make it clear that events depend on the parameter key k, we tagged them with the key. Without any knowledge about the semantics of the program to monitor, any monitor for this property *must* store all the authenticated keys, i.e., all the instances of the parameter k. Indeed, without that, there is no way to know whether a key instance has been authenticated or not when a use event is observed on that key. The number of such key instances is theoretically unbounded, so, in the worst case, any monitor for this property can be incrementally slower and eventually run out of resources.

As seen in Section 9, the runtime overhead due to opaque monitoring of parametric properties tends to be manageable in practice. By "opaque" we mean that no semantic information about the source code of the monitored program is used. If the lack of an

efficiency guarantee is a problem in some applications, then the alternative is to statically analyze the monitored program and to use the obtained semantic information to eliminate the need for monitoring. For example, static analyses like those in [25, 10, 12, 18] may significantly reduce the need for instrumentation, even eliminate it completely. Moreover, model-checking techniques for parametric properties could also be used for actually proving that the properties hold and thus they need not be monitored; however, we are not aware of model checking approaches to verifying parametric properties as presented in this paper.

9. Implementation in JavaMOP and RV

The discussed parametric monitoring technique is now fully implemented in two runtime verification systems, namely in JavaMOP (see `http://javamop.org`) and in RV [27] (developed by a startup company, Runtime Verification, Inc.; the RV system is currently publicly unavailable – contact the first author for an NDA-protected version of RV). Here we first informally discuss several optimizations implemented in the two runtime verification systems, and then we discuss our experiments and the evaluation of the two systems.

9.1. **Implementation Optimizations.** Both JavaMOP and RV apply several optimizations to the algorithm $\mathbb{C}\langle X \rangle$ in Section 8.2, to reduce its runtime overhead. These are not discussed in depth here, because they are orthogonal to the main objective of this paper.

9.1.1. *Optimizations in JavaMOP.* Note that $\mathbb{C}\langle X \rangle$ 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. Even though our semantics and theoretical algorithms for parametric monitoring in this paper work with infinite sets of parameters, our current implementation in JavaMOP assumes that the set of parameters X is bounded and fixed apriori (declared as part of the specification to monitor). A simple analysis of the events appearing in the specification allows 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 static analysis of the specification, discussed in [13, 10], exhaustively explores all possible event combinations that can lead to situations of interest to the property, such as to violation, validation, etc. Such information is useful to reduce the number of loop iterations by skipping iterations over parameter instances that cannot affect the result of monitoring. These static analyses are currently used at compile time in our new JavaMOP implementation 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 convenient to start the monitoring process only when certain events are received. Such events are called monitor creation events in [14]. 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 old JavaMOP [14] supported only traces whose monitoring started with a fully instantiated monitor creation event; this was perceived (and admitted) as a performance-tradeoff limitation of JavaMOP [4] (and [14]). Interestingly, it now becomes just a common-case optimization of our novel, general and unrestricted technique presented here.

9.1.2. *Optimizations in RV.* The RV system implements all the optimizations in JavaMOP and adds two other important optimizations that significantly reduce the overhead.

The first additional optimization of RV is a non-trivial garbage-collector [20]. Note that JavaMOP also has a garbage collector, but it only does the obvious: it garbage collects a monitor instance only when all its corresponding parameter instances are collected. RV performs a static analysis of the property to monitor and, based on that, it garbage collects a monitor instance as soon as it realizes that it can never trigger in the future. This can happen when any triggering behavior needs at least one event that can only be generated in the presence of a parameter instance that is already dead. Consider, for example, the safe iterator example in Section 2.3, and consider that iterator i_7 is created for collection c_3. Then a monitor instance corresponding to the parameter instance $\langle c_3\ i_7 \rangle$ is created and manged. Suppose that, at some moment, the iterator i_7 is garbage collected by the JVM. Can the monitor instance corresponding to $\langle c_3\ i_7 \rangle$ be garbage collected? Not in JavaMOP, because, for safety, JavaMOP collects a monitor only when all its parameter instances are collected, and in this case c_3 is still alive. However, this monitor is flagged for garbage collection in RV. The rationale for doing so is that the only way for the monitor to trigger is to eventually encounter a next event with i_7 as parameter, but that event can never be generated because i_7 is dead. Note, on the other hand, that the monitor $\langle c_3\ i_7 \rangle$ cannot be garbage collected if c_3 is collected but i_7 is still alive, because the iterator alone can still violate the safe-iterator property, even if its corresponding collection is already dead.

Runtime verification systems like JavaMOP and Tracematches use off-the-shelf weak reference libraries to implement their garbage collectors. However, it turns out that these libraries, in order to be general and thus serve their purpose, perform many checks that are unnecessary in the context of monitoring. The second optimization of RV in addition to those of JavaMOP consists of a collection of data structures based on weak references, which was carefully engineered to take full advantage of the particularities of monitoring parametric properties. These data structures allow for effective indexing and lazy collection of monitors, to minimize the number of expensive traversals of the entire pool of monitors. Moreover, RV caches monitor instances to save time when the same monitor instances are accessed frequently. For example, there is a high chance that the same iterator is accessed several times consecutively by a program, in which case saving and then retrieving the same corresponding monitor instances from the data-structures at each iterator access can take considerable unnecessary overhead.

9.2. **Experiments and Evaluation.** We next discuss our experience with using the two runtime verification systems that implement optimized variants of the parametric property monitoring techniques describe in this paper. Also, we compare their performance with that of Tracematches, which is, at our knowledge, the most efficient runtime verification system besides JavaMOP and RV. Recall that Tracematches achieves virtually the same semantics of parametric monitoring like ours, but using a considerably different approach.

9.2.1. *Experimental Settings.* We used a Pentium 4 2.66GHz / 2GB RAM / Ubuntu 9.10 machine and version 9.12 of the DaCapo (DaCapo 9.12) benchmark suite [8], the most up-to-date version. We also present some of the results of our experiments using the previous version of DaCapo, 2006-10 MR2 (DaCapo 2006-10), namely those for the bloat and jython benchmarks. DaCapo 9.12 does not provide the bloat benchmark from the DaCapo 2006-10, which we favor because it generates large overheads when monitoring iterator-based properties. The bloat benchmark with the UNSAFEITER specification causes 19194% runtime overhead (i.e., 192 times slower) and uses 7.7MB of heap memory in Tracematches, and causes 569% runtime overhead and uses 147MB in JavaMOP, while the original program uses only 4.9MB. Also, although the DaCapo 9.12 provides jython, Tracematches cannot instrument jython due to an error that we were not able to understand or fix. Thus, we present the result of jython from the DaCapo 2006-10. The default data input for DaCapo was used and the -converge option to obtain the numbers after convergence within ±3%. Instrumentation introduces a different garbage collection behavior in the monitored program, sometimes causing the program to slightly outperform the original program; this accounts for the negative overheads seen in both runtime and memory.

We used the Sun JVM 1.6.0 for the entire evaluation. The AspectJ compiler (ajc) version 1.6.4 is used for weaving the aspects generated by JavaMOP and RV into the target benchmarks. Another AspectJ compiler, abc [3] 1.3.0, is used for weaving Tracematches properties because Tracematches is part of abc and does not work with ajc. For JavaMOP, we used the most recent release version, 2.1.2. For Tracematches, we used the most recent release version, 1.3.0, from [36], which is included in the abc compiler as an extension. To figure out the reason that some examples do not terminate when using Tracematches, we also used the abc compiler for weaving aspects generated by JavaMOP and RV. Note that JavaMOP and RV are AspectJ compiler-independent. They show similar overheads and terminate on all examples when using the abc compiler for weaving as when ajc is used. Because the overheads are similar, we do not present the results of using abc to weave JavaMOP and RV generated aspects in this paper. However, using abc to weave JavaMOP and RV properties confirms that the high overhead and non-termination come from Tracematches itself, not from the abc compiler.

The following properties are used in our experiments. Some of them were already discussed in Sections 1.1 and 2, others are borrowed from [10, 11, 26, 13].

- HASNEXT: Do not use the next element in an iterator without checking for the existence of it;

- UNSAFEITER: Do not update a collection when using the iterator interface to iterate its elements;

- UNSAFEMAPITER: Do not update a map when using the iterator interface to iterate its values or its keys;

- UNSAFESYNCCOLL: If a collection is synchronized, then its iterator also should be accessed synchronously;

- UNSAFESYNCMAP: If a collection is synchronized, then its iterators on values and keys also should be accessed synchronized.

	ORIG (sec)	HasNext			UnsafeIter			UnsafeMapIter			UnsafeSyncColl			UnsafeSyncMap			ALL
(A)		TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	RV
bloat	3.6	2119	448	116	19194	569	251	∞	1203	178	1359	746	212	1942	716	130	982
jython	8.9	13	0	0	11	0	1	150	18	3	11	1	1	10	0	0	4
avrora	13.6	45	54	55	637	311	118	∞	113	42	75	144	80	54	74	16	275
batik	3.5	3	2	3	355	9	8	∞	8	5	208	9	9	5	3	0	28
eclipse	79.0	-2	4	-1	0	-1	-1	5	-3	0	-4	2	1	∞	-1	-1	0
fop	2.0	200	49	48	350	21	13	∞	58	14	∞	78	25	∞	71	19	133
h2	18.7	89	17	13	128	9	4	1350	21	6	868	21	4	83	20	5	23
luindex	2.9	0	0	1	0	0	1	1	4	1	1	1	1	2	0	0	1
lusearch	25.3	-1	1	0	1	2	2	2	2	0	4	0	1	3	1	1	3
pmd	8.3	176	84	59	1423	162	123	∞	571	188	1818	192	76	∞	144	26	620
sunflow	32.7	47	5	3	7	2	0	9	4	1	13	6	5	17	6	6	6
tomcat	13.8	8	1	1	37	1	1	3	1	1	2	0	1	2	1	3	1
tradebeans	45.5	0	-1	1	1	1	2	5	3	-1	-1	1	2	3	1	5	2
tradesoap	94.4	1	3	0	2	1	1	2	0	1	0	0	1	2	2	5	1
xalan	20.3	4	2	2	27	7	2	10	5	2	3	2	3	4	4	3	4
(B)		TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	RV
bloat	4.9	56.8	19.3	13.2	7.7	146.8	79.0	∞	173.4	56.1	6.8	127.9	48.3	6.9	55.4	12.7	340.9
jython	5.3	5.7	4.6	4.8	4.9	4.6	4.8	6.0	19.5	4.7	5.3	4.5	4.4	5.9	4.8	5.1	4.7
avrora	4.7	4.6	12.4	9.1	4.4	136.2	15.8	∞	14.7	8.5	4.3	28.0	12.6	4.4	13.0	4.9	22.3
batik	79.1	79.2	78.7	79.3	75.2	93.6	86.6	∞	91.2	79.6	78.2	93.2	85.1	79.9	86.9	76.7	104.3
eclipse	95.9	100.8	107.6	97.1	98.3	100.0	110.3	106.9	93.8	101.1	100.4	109.2	90.1	∞	98.6	98.7	98.9
fop	20.7	97.4	47.1	52.5	24.3	24.2	29.4	∞	69.2	28.1	∞	54.8	24.8	∞	55.9	25.2	47.5
h2	265.0	267.8	598.5	565.2	267.2	266.2	262.4	312.4	688.3	268.2	271.4	690.3	265.5	271.0	718.3	270.0	283.7
luindex	6.8	5.6	5.5	5.6	6.3	6.9	6.8	7.4	8.2	6.9	7.4	7.4	7.5	7.1	7.4	11.0	11.8
lusearch	4.6	4.7	4.4	4.8	4.6	4.8	4.2	4.0	4.3	4.8	4.5	4.5	4.6	4.6	4.8	4.7	4.7
pmd	18.0	56.9	59.8	48.5	17.2	146.3	86.4	∞	212.7	93.6	20.3	238.4	84.6	∞	117.1	32.9	420.0
sunflow	4.4	4.5	4.8	4.9	4.8	4.3	4.7	4.7	4.4	4.4	5.1	4.3	4.9	4.5	4.7	4.5	4.6
tomcat	11.6	11.4	12.3	11.4	12.5	11.0	11.5	11.9	11.4	11.0	11.3	11.3	11.3	11.4	11.4	11.8	11.8
tradebeans	63.2	62.9	62.7	62.1	63.7	63.9	64.1	63.3	62.5	62.7	63.2	62.8	62.0	64.0	62.8	64.0	62.5
tradesoap	64.1	61.8	62.3	63.3	63.4	63.1	64.4	64.1	63.5	62.0	60.7	65.0	65.9	65.5	64.5	65.6	64.5
xalan	4.9	4.9	5.0	5.1	4.9	4.9	4.9	4.9	4.5	4.9	5.0	4.8	5.0	5.1	4.9	4.9	5.0

Figure 5: Comparison of Tracematches (TM), JavaMOP (MOP), and RV: (A) average *percent* runtime overhead; (B) total peak memory usage in MB. (convergence within 3%, ∞: not terminated after 1 hour)

	HasNext				UnsafeIter				UnsafeMapIter				UnsafeSyncColl				UnsafeSyncMap			
	E	M	FM	CM	E	M	FM	CM	E	M	FM	CM	E	M	FM	CM	E	M	FM	CM
bloat	155M	1.9M	1.9M	1.8M	81M	1.9M	1.8M	1.6M	74M	3.6M	43K	3.4M	143M	4.1M	0	3.7M	161M	3.4M	0	3.4M
jython	106	50	47	26	179K	50	38	38	179K	101K	94	101K	156	100	0	83	256	150	0	122
avrora	1.5M	909K	850K	765K	1.4M	909K	860K	808K	1.3M	1.2M	18	1.2M	2.4M	1.8M	0	1.7M	1.5M	909K	0	904K
batik	49K	24K	21K	21K	125K	24K	21K	10K	55K	33K	140	27K	73K	50K	0	34K	50K	26K	0	26K
eclipse	226K	7.6K	5.3K	2.9K	119K	6.6K	5.1K	2.6K	113K	22K	2.2K	7.8K	233K	15K	0	7.5K	241K	18K	0	9.2K
fop	1.0M	184K	74K	151K	709K	7.7K	7.2K	1.8K	499K	177K	67	160K	1.2M	239K	0	217K	1.2M	231K	0	213K
h2	27M	6.5M	6.0M	5.6M	12M	3.7K	3.3K	1.3K	12M	6.6M	9	6.5M	27M	6.5M	0	6.5M	27M	6.5M	0	6.5M
luindex	371	66	40	2	4.4K	65	39	0	378	183	2	59	436	132	0	30	472	125	0	25
lusearch	1.4K	131	196	114	748K	130	210	18	20K	944	338	1.4K	1.7K	262	0	402	1.8K	263	0	158
pmd	8.3M	789K	694K	571K	6.4M	551K	473K	382K	4.3M	1.3M	110K	1.1M	8.8M	1.5M	0	1.3M	8.6M	1.1M	0	999K
sunflow	2.7M	101K	101K	100K	1.3M	2	0	0	1.3M	83K	0	83K	2.7M	101K	0	101K	2.7M	101K	0	101K
tomcat	25	6	0	0	132	4	0	0	68	26	0	0	29	10	0	0	33	12	0	0
tradebeans	11	3	0	0	31	2	0	0	29	13	0	0	13	5	0	0	15	6	0	0
tradesoap	11	3	0	0	31	2	0	0	29	13	0	0	13	5	0	0	15	6	0	0
xalan	11	3	0	0	8.9K	2	0	0	119K	20K	0	20K	13	5	0	0	15	6	0	0

Figure 6: Monitoring statistics: number of events (E), number of created monitors (M), number of flagged monitors (FM), number of collected monitors (CM).

All of them are tested on Tracematches, JavaMOP, and RV for comparison. We also monitored all five properties at the same time in RV, which was not possible in other monitoring systems for performance reasons or structural limitations.

9.2.2. *Results and Discussions.* Figures 5 and 6 summarize the results of the evaluation. Note that the structure of the DaCapo 9.12 allows us to instrument all of the benchmarks plus all supplementary libraries that the benchmarks use, which was not possible for Da-Capo 2006-10. Therefore, fop and pmd show higher overheads than the benchmarks using DaCapo 2006-10 from [13]. While other benchmarks show overheads less than 80% in JavaMOP, bloat, avrora, and pmd show prohibitive overhead in both runtime and memory performance. This is because they generate many iterators and all properties in this evaluation are intended to monitor iterators. For example, bloat creates 1,625,770 collections and 941,466 iterators in total while 19,605 iterators coexist at the same time at peak, in an execution. avrora and pmd also create many collections and iterators. Also, they call hasNext() 78,451,585 times, 1,158,152 times and 4,670,555 times and next() 77,666,243 times, 352,697 times and 3,607,164 times, respectively. Therefore, we mainly discuss those three examples in this section, although RV shows improvements for other examples as well.

Figure 5 (A) shows the percent runtime overhead of Tracematches, JavaMOP, and RV. Overall, RV averages two times less runtime overhead than JavaMOP and orders of magnitude less runtime overhead than Tracematches (recall that these are the most optimized runtime verification systems). With bloat, RV shows less than 260% runtime overhead for each property, while JavaMOP always shows over 440% runtime overhead and Tracematches always shows over 1350% for completed runs and *crashed* for UNSAFEMAPITER. With avrora, on average, RV shows 62% runtime overhead, while JavaMOP shows 139% runtime overhead and Tracematches shows 203% and hangs for UNSAFEMAPITER. With pmd, on average, RV shows 94% runtime overhead, while JavaMOP shows 231% runtime overhead and Tracematches shows 1139% and hangs for UNSAFEMAPITER and UNSAFESYNCMAP.

Also, RV was tested with all five properties together and showed 982%, 275%, and 620% overhead, respectively, which are still faster or comparable to monitoring one of many properties alone in JavaMOP or Tracematches. The overhead for monitoring all the properties simultaneously can be slightly larger than the sum of their individual overheads since the additional memory pressure makes the JVM's garbage collection behave differently.

Figure 5 (B) shows the peak memory usage of the three systems. RV has lower peak memory usage than JavaMOP in most cases. The cases where RV does not show lower peak memory usage are within the limits of expected memory jitter. However, memory usage of RV is still higher than the memory usage of Tracematches in some cases. Tracematches has several finite automata specific memory optimizations [4], which cannot be implemented in formalism-independent systems like RV and JavaMOP. Although Tracematches is sometimes more memory efficient, it shows prohibitive runtime overhead monitoring bloat and pmd. There is a trade-off between memory usage and runtime overhead. If RV more actively removes terminated monitors, memory usage will be lower, at the cost of runtime performance. Overall, the monitor garbage collection optimization in RV achieves the most efficient parametric monitoring system with reasonable memory performance.

Figure 6 shows the number of triggered events, of created monitors, of monitors flagged as unnecessary by RV's optimization, and of monitors collected by the JVM. Among the DaCapo examples, bloat, avrora, h2, pmd and sunflow generated a very large number of events (millions) in all properties, resulting in millions of monitors created in most cases. h2 does not exhibit large overhead because monitor instances in h2 have shorter lifetimes, therefore the created monitor instances are not used heavily like in bloat. sunflow has millions of events but does not create as many monitor instances as as other benchmarks. When

monitoring the HasNext and UnsafeIter properties, RV's garbage collector effectively flagged monitors as unnecessary and most were collected by the JVM.

The experimental evaluation in this section shows that the approach to parametric trace slicing and monitoring discussed in this paper is indeed feasible, provided that it is not implemented naively. Indeed, as seen in the tables in this section, implementation optimizations make a huge difference in the runtime and memory overhead. This paper was not dedicated to optimizations and implementations; its objective was to only introduce the mathematical notions, notations, proofs and abstract algorithms underlying the semantical foundation of parametric properties and their monitoring. Current and future implementations are and will build on this foundation, applying specific optimizations and heuristics to reduce the runtime or the memory overhead caused by monitoring.

10. Concluding Remarks, Future Work and Acknowledgments

A semantic foundation for parametric traces, properties and monitoring was proposed. A parametric trace slicing technique, which was discussed and proved correct, allows the extraction of all the non-parametric trace slices from a parametric slice by traversing the original trace only once and dispatching each parametric event to its corresponding slices. It thus enables the leveraging of any non-parametric, i.e., conventional, trace analysis techniques to the parametric case. A parametric monitoring technique, also discussed and proved correct, makes use of it to monitor arbitrary parametric properties against parametric execution traces using and indexing ordinary monitors for the base, non-parametric property. Optimized implementations of the discussed techniques in JavaMOP and RV reveal that their generality, compared to the existing similar but ad hoc and limited techniques in current use, does not come at a performance expense. Moreover, further static analysis optimizations like those in [25, 10, 12, 18] may significantly reduce the runtime and memory overheads of monitoring parametric properties based on the techniques and algorithms discussed in this paper.

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. Preliminary steps in this direction are reported in [22].

Sadly, the second author, Feng Chen, passed away on August 8, 2009, in the middle of this project, due to an undetected blood clot. Feng was the main developer of JavaMOP and a co-inventor of most of its underlying techniques and algorithms, including those in this paper. His results and legacy will outlive him. May his soul rest in peace.

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, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. ABC: an extensible AspectJ compiler. In *Aspect-Oriented Software Development (AOSD'05)*, pages 87–98. ACM, 2005.

[4] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitoring feasible. In R. P. Gabriel, editor, *OOPSLA'07*. ACM, 2007.

[5] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, volume 2937 of *LNCS*, pages 44–57, 2004.

[6] H. Barringer, D. E. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: From Eagle to RuleR. In *Runtime Verification (RV'07)*, volume 4839 of *LNCS*, pages 111–125, 2007.

[7] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20, 2011.

[8] 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.

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

[10] E. Bodden, F. Chen, and G. Roşu. Dependent advice: A general approach to optimizing history-based aspects. In *AOSD'09*, pages 3–14. ACM, 2009.

[11] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *European Conference on Object Oriented Programming (ECOOP'07)*, volume 4609 of *LNCS*, pages 525–549. Springer, 2007.

[12] E. Bodden, P. Lam, and L. Hendren. Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In *Runtime Verification (RV'10)*, volume 6418 of *LNCS*, pages 183–197. Springer, 2010.

[13] F. Chen, P. Meredith, D. Jin, and G. Rosu. Efficient formalism-independent monitoring of parametric properties. In *ASE'09*. IEEE/ACM, 2009.

[14] F. Chen and G. Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *OOPSLA'07*, pages 569–588. ACM, 2007.

[15] F. Chen and G. Roşu. Mining Parametric State-Based Specifications from Executions. Technical Report UIUCDCS-R-2008-3000, Dept. of Computer Science at UIUC, 2008.

[16] F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *TACAS'09*, volume 5505 of *LNCS*, pages 246–261, 2009.

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

[18] M. Dwyer, R. Purandare, and S. Person. Runtime verification in context: Can optimizing error detection improve fault diagnosis. In *Runtime Verification (RV'10)*, volume 6418 of *LNCS*, pages 36–50. Springer, 2010.

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

[20] D. Jin, P. O. Meredith, D. Griffith, and G. Roşu. Garbage collection for monitoring parametric properties. In *Programming Language Design and Implementation (PLDI'11)*, pages 415–424. ACM, 2011.

[21] O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Form. Methods Syst. Des.*, 19:291–314, October 2001.

[22] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *Proceeding of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 591–600. ACM, 2011.

[23] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[24] S. Maoz and D. Harel. From multi-modal scenarios to code: compiling lscs into aspectj. In *FSE'06*, pages 219–230, 2006.

[25] 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.

[26] P. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. In *ASE'08*. IEEE/ACM, 2008.

[27] P. Meredith and G. Roşu. Runtime verification with the RV system. In *First International Conference on Runtime Verification (RV'10)*, volume 6418 of *Lecture Notes in Computer Science*, pages 136–152. Springer, 2010.

[28] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the mop runtime verification framework. *Journal on Software Tools for Technology Transfer (J. of STTT)*, 2010. To appear.

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

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

[31] G. Roşu, F. Chen, and T. Ball. Synthesizing monitors for safety properties – this time with calls and returns –. In *Runtime Verification (RV'08)*, volume 5289 of *LNCS*, pages 51–68, 2008.

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

[33] V. Stolz. Temporal assertions with parameterized propositions. In O. Sokolsky and S. Tasiran, editors, *Runtime Verification*, volume 4839 of *Lecture Notes in Computer Science*, pages 176–187. Springer Berlin / Heidelberg, 2007.

[34] R. E. Strom and S. Yemeni. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, January 1986.

[35] D. Tabakov and M. Vardi. Optimized temporal monitors for systemc. In *First International Conference on Runtime Verification (RV'10)*, volume 6418 of *Lecture Notes in Computer Science*. Springer, 2010.

[36] Tracematches Benchmarks. http://abc.comlab.ox.ac.uk/tmahead.