Efficient Formalism-Independent Monitoring of Parametric Properties *

Feng Chen Patrick O'Neil Meredith Dongyun Jin Grigore Roşu
University of Illinois at Urbana-Champaign
{fengchen,pmeredit,djin3,grosu}@illinois.edu

Abstract

Parametric properties provide an effective and natural means to describe object-oriented system behaviors, where the parameters are typed by classes and bound to object instances at runtime. Efficient monitoring of parametric properties, in spite of increasingly growing interest due to applications such as testing and security, imposes a highly non-trivial challenge on monitoring approaches due to the potentially huge number of parameter instances. Existing solutions usually compromise their expressiveness for performance or vice versa. In this paper, we propose a generic, in terms of specification formalism, yet efficient, solution to monitoring parametric specifications. Our approach is based on a general algorithm for slicing parametric traces and makes use of *static* knowledge about the desired property to optimize monitoring. The needed knowledge is not specific to the underlying formalism and can be easily computed when generating monitoring code from the property. Our approach works with any specification formalism, providing better and extensible expressiveness. Also, an thorough evaluation shows that our technique out performs other state-of-art techniques optimized for particular logics or properties.

1. Introduction

Monitoring executions of a system against expected properties plays an important role not only in different stages of software development, e.g., testing and debugging, but also in the deployed system as a mechanism to increase system reliability. Numerous approaches, such as (12; 14; 10; 7; 3; 1; 2; 15; 11; 8), have been proposed to build effective and efficient monitoring solutions for different applications. More recently, monitoring of parametric specifications, i.e., specifications with free variables, has received increasing interest due to its effectiveness at capturing system behaviors, as shown in the following example about interaction between the classes Map, Collection and Iterator in Java.

Map and Collection implement data structures for mappings and collections, respectively. Iterator is an interface used to enumerate elements in a collection-typed object. One can also enumerate elements in a Map object using Iterator. But, since a Map object contains key-value pairs, one needs to first obtain a collection object that represents the contents of the map, e.g., the set of keys or the set of values stored in the map, and then create an iterator from the obtained collection. An intricate safety property in this usage, according to the Java API specification, is that when the iterator is used to enumerate elements in the map, the contents of the map should not be changed, or unexpected behaviors may occur. A *violating* behavior with regards to this property, which we call UnsafeMapIterator, can be naturally specified using future time linear temporal logic (FTLTL) with parameters: given that m, c, i are objects of Map, Collection and Iterator, respectively, $\forall m, c, i. \diamond (\mathsf{create\_coll}\langle m, c\rangle \wedge \diamond(\mathsf{create\_iter}\langle c, i\rangle \wedge \diamond(\mathsf{update\_map}\langle m\rangle \wedge \diamond \mathsf{use\_iter}\langle i\rangle)))$, where create_coll is creating a collection from a map, create_iter is creating an iterator from a collection, update_map is updating the map, and use_iter is using the iterator; $\diamond$ means eventually in the future. The formula describes the following sequence of actions: Collection c is obtained from a Map m, an iterator i is created from c, m is changed, and then i is accessed. When an observed execution satisfies this formula, the UnsafeMapIterator property is broken in the execution.

It is highly non-trivial to monitor such parametric specifications efficiently. We may see a tremendous number of parameter instances during the execution; for example, it is not uncommon to see hundreds of thousands of iterators in one execution. Also, some events may contain partial information about parameters, making it more difficult in locating other relevant parameter bindings during the monitoring process; for example, in the above specification, when a up-

* Supported in part by NSF grants CCF-0448501, CNS-0509321 and CNS-0720512, by NASA contract NNL08AA23C, by the Microsoft/Intel funded Universal Parallel Computing Research Center at UIUC, and by several Microsoft gifts.

date_map$\langle m \rangle$ is received, we need to find all create_coll$\langle m, c \rangle$ events with the same binding for m, and transitively, all create_iter$\langle c, i \rangle$ with the same c as that create_coll.

Several approaches were introduced to support the monitoring of parametric specifications, including Eagle (3), Tracematches (1; 2), PQL (15), PTQL (11) and MOP (8). However, they are all limited in terms of supported specification formalisms or viable execution traces. Most techniques, e.g., Eagle, Tracematches, PQL and PTQL, follow a formalism-dependent approach, that is, they have their parametric specification formalisms hardwired, e.g., regular patterns (like Tracematches), context-free patterns (like PQL) with parameters, etc., and then develop algorithms to generate monitoring code for the particular formalisms. Although this approach provides a feasible solution to monitoring parametric specifications, we argue that it not only has limited expressiveness, but also causes unnecessary complexity in developing optimal monitor generation algorithms, often leading to inefficient monitoring. In fact, experiments in (8) and Section 7 show that our formalism-independent solution generates more efficient monitoring code than other existing tools. MOP, on the other hand, does not fix the formalism to use in the specification. Instead, MOP provides a generic framework for monitoring of parametric specifications, which allows one to use existing non-parametric formalisms in parametric specifications. Unfortunately, however, the original MOP algorithm (8) for parametric monitoring supports only those specifications in which the first event for any matching trace instantiates all the parameters of the property. This limitation prevents it from monitoring a large subset of parametric properties, including the above UnsafeMapIterator property: the traces specified by UnsafeMapIterator begin with create_coll, which does not instantiate parameter i.

In this paper, we present a general technique to build optimized parametric monitors from non-parametric monitors, following the spirit of MOP but *without limitation*. The presented technique is based on the theoretical results in (9), which was focused on a general, theoretical solution for handling parametric trace and proposed a conceptual algorithm[1]. In this novel technique, we apply knowledge about the monitored *property* to improve efficiency. The needed knowledge, encoded as *enable sets*, depends only on the property and not on the formalism in which it is specified. It can be easily computed as a *side effect* when generating a monitor from the property, as discussed in Section 5.

Our technique has been implemented in the latest version of JavaMOP[2]. An extensive evaluation shows that the proposed technique not only allows for greater expressiveness, but also significantly improves the efficiency of monitoring in comparison to prior techniques with fixed logical formalisms. This new technique of optimization based on enable sets, combined with the new general parametric algorithm from (9), represents the first *efficient, modular* technique for monitoring fully general properties (i.e., the properties do not need to instantiate all the parameters in the creation events or use a fixed logical formalism). In fact, it is *more* efficient than the systems that do use a fixed formalism. For the (enable-set-)optimized JavaMOP, only 7 out of 66 of our tested cases caused more than 10% runtime overhead. The numbers for the non-optimized JavaMOP and Tracematches are 9 out of 66 and 15 out of 44, respectively. On two cases the optimized JavaMOP has over an order of magnitude less overhead than Tracematches, and the non-optimized JavaMOP fails to complete the runs, running out of memory. On five other cases, Tracematches has at least twice the overhead of optimized JavaMOP. On any case with noticeable overhead, the enable set optimization produces a notable reduction in overhead.

Contributions. The major contributions of this paper are:

1. A formalism-independent technique for monitoring parametric properties, which overcomes the limitations of existing techniques without reducing performance.
2. A novel concept of enable sets, which encodes static knowledge of the property to monitor and facilitates the optimization of the monitoring process, together with algorithms to compute enable sets for several requirements specification formalisms.
3. An extensive evaluation and comparison of the proposed solution with Tracematches, an efficient monitoring system for regular expression based properties (1; 2).

Outline. The remainder of this paper is as follows: Section 2 provides an intuitive overview of our technique, providing an understanding of the more formal parts of the paper that follow. Section 3 provides definitions and examples regarding parametric traces. Section 4 discusses modification of the online monitoring algorithm presented in (9) to only create monitor instances for events deemed *monitor creation events*. Section 5 presents the enable sets, which drive our optimization. Section 6 presents the finished, optimized online trace algorithm, with a proof of correctness. Section 7 discusses the implementation of the algorithm, and presents the results of our evaluation, which show the efficiency of our technique. Lastly, Section 8 concludes the paper.

2. Approach Overview

To illustrate our technique we expand the UnsafeMapIterator example discussed in Section 1. Figure 1 shows a JavaMOP specification of the UnsafeMapIterator property using five different formalisms: finite state machines (FSM), extended regular expressions (ERE), context-free grammars (CFG), future-time linear temporal logic (FTLTL), and past-time linear temporal logic (PTLTL). Because each of the properties

[1] Note that the evaluation results in (9), are based on the technique presented in this paper. The technique was only very briefly mentioned in (9), due to its different focus, and because the optimization had not yet been formalized.

[2] JavaMOP is the Java specialization of MOP, which is itself a framework generic in requirements specification formalisms (8).

```
UnsafeMapIterator(Map m, Collection c, Iterator i){
   event create_coll after(Map m) returning(Collection c) : (call(* Map.values()) || call(* Map.keySet())) && target(m) {}
   event create_iter after(Collection c) returning(Iterator i) : call(* Collection+.iterator()) && target(c) {}
   event use_iter before(Iterator i) : call(* Iterator+.next()) && target(i) {}
   event update_map after(Map m) : (call(* Map.remove*(..)) || call(* Map.put*(..))
      || call(* Map.putAll*(..)) || call(* Map.clear())) && target(m) {}

   fsm: start [ create_coll -> s1 ]
        s1    [ update_map -> s1, create_iter -> s2 ]
        s2    [ use_iter -> s2,   update_map -> s3  ]
        s3    [ update_map -> s3, use_iter -> end   ]
        end   []
      @end{ System.out.println("fsm: Accessed Invalid Iterator!"); __RESET; }

   ere : create_coll update_map* create_iter use_iter* update_map update_map* use_iter
      @match{ System.out.println("ere: Accessed Invalid Iterator!"); __RESET; }

   cfg : S -> create_coll Updates create_iter Nexts update_map Updates use_iter,
         Nexts -> Nexts use_iter | epsilon,
         Updates -> Updates update_map | epsilon
      @match{ System.out.println("cfg: Accessed Invalid Iterator!"); __RESET;}

   ftltl: <>(create_coll /\ <> (create_iter /\ <> (update_map /\ <> use_iter)))
      @validation{ System.out.println("ftltl: Accessed Invalid Iterator!"); __RESET; }

   ptltl: use_iter -> ((<*> (create_iter /\ (<*> create_coll))) -> ((!update_map) Since create_iter))
      @violation{ System.out.println("ptltl: Accessed Invalid Iterator!"); __RESET; }
}
```

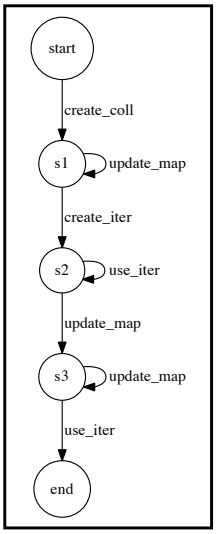

Figure 1. FSM, ERE, CFG, FTLTL, and PTLTL UnsafeMapIterator. Inset: Graphical Depiction of the Property.

in Figure 1 is the same, five messages will be reported whenever an Iterator is incorrectly used after an update to the underlying Map. We show all five of them to emphasize the formalism-independence of our approach. On the first line, we name the specified property and give the parameters used in the specification. Then we define the involved events using the AspectJ (13) syntax. For example, create_coll is defined as the return value of functions values and keyset of Map. We adopt AspectJ syntax to define events in JavaMOP because it is an expressive language for defining observation points in a Java program. As mentioned, every event may instantiate some parameters at runtime. This can be seen in Figure 1: create_coll will instantiate parameters m and c using the target and the return value of the method call. When one defines a pattern or formula there are implicit events, which must begin traces; we call these *monitor creation* events. For example, in a pattern language like ERE, the monitor creation events are the first events that appear in the pattern. We assume a semantics where events that occur before monitor creation events are ignored. The crucial point is that this example could not be monitored using the original MOP parametric monitoring algorithm (8) because create_coll, the only monitor creation event, does not instantiate the Iterator parameter i.

JavaMOP automatically synthesizes AspectJ instrumentation code from the specification, which is weaved into the program we wish to monitor by any standard AspectJ compiler. In this way, executions of the monitored program will produce traces made up of events defined in the specification, as those in Figure 1. Consider the example eleven event trace in Figure 2 over the events defined in Figure 1. The # column gives the numbering of the events for easy reference. Every event in the trace starts with the name of the event, e.g., create_coll, followed by the parameter binding information, e.g., $\langle m_1, c_1 \rangle$ that binds parameters m and c with a map object m_1 and a collection c_1, respectively. Such a trace is called a *parametric trace* since it contains events with parameters.

#	Event	#	Event
1	create_coll$\langle m_1, c_1 \rangle$	7	update_map$\langle m_1 \rangle$
2	create_coll$\langle m_1, c_2 \rangle$	8	use_iter$\langle i_2 \rangle$
3	create_iter$\langle c_1, i_1 \rangle$	9	create_coll$\langle m_2, c_3 \rangle$
4	create_iter$\langle c_1, i_2 \rangle$	10	create_iter$\langle c_3, i_4 \rangle$
5	use_iter$\langle i_1 \rangle$	11	use_iter$\langle i_4 \rangle$
6	create_iter$\langle c_2, i_3 \rangle$		

Figure 2. Possible Execution Trace Over the Events Specified in UnsafeMapIterator.

Our approach to monitoring parametric traces against parametric properties is based on the observation that each parametric trace actually contains multiple *non-parametric trace slices*, each for a particular parameter binding instance. The formal definition of the trace slice can be found in Section 3, but intuitively, a slice of a parametric trace for a particular parameter binding consists of names of all the events that have *less informative* parameter bindings. Informally, a parameter binding b_1 is less informative than a parameter binding b_2 if and only if the parameters for which they have bindings agree, and b_2 binds either an equal number of parameters or more parameters: parameter $\langle m_1, c_2 \rangle$ is less informative than $\langle m_1, c_2, i_3 \rangle$ because the parameters they both bind, m and c, agree on their values, m_1 and c_2, respectively, and $\langle m_1, c_2, i_3 \rangle$ binds one more parameter. Figure 3 shows the trace slices and their corresponding parameter bindings contained in the trace in Figure 2. The Status column denotes the output category that the slice falls into (for ERE). In

Instance	Slice	Status
$\langle m_1 \rangle$	update_map	?
$\langle m_1, c_1 \rangle$	create_coll update_map	?
$\langle m_1, c_2 \rangle$	create_coll update_map	?
$\langle m_2, c_3 \rangle$	create_coll	?
$\langle m_1, c_1, i_1 \rangle$	create_coll create_iter use_iter update_map	?
$\langle m_1, c_1, i_2 \rangle$	create_coll create_iter update_map use_iter	match
$\langle m_1, c_2, i_3 \rangle$	create_coll create_iter update_map	?
$\langle m_2, c_3, i_4 \rangle$	create_coll create_iter use_iter	?

Figure 3. Slices for the Trace in Figure 2.

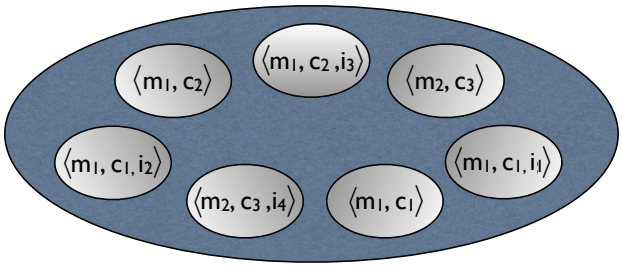

Figure 4. A Monitor Set (Parametric Monitor) with Corresponding Parameter Instance Monitors.

this case everything but the slice for $\langle m_1, c_1, i_2 \rangle$, which matches the property, is in the "?" (undecided) category. For example, the trace for the binding $\langle m_1, c_1 \rangle$ contains create_coll update_map (the first and seventh events in the trace) and the trace for the binding $\langle m_1, c_1, i_2 \rangle$ is create_coll create_iter update_map use_iter (the first, fourth, seventh, and eighth events in the trace).

Based on this observation, our approach creates a set of monitor instances during the monitoring process, each handling a trace slice for a parameter binding. Figure 4 shows the set of monitors created for the trace in Figure 2, each monitor labeled by the corresponding parameter binding. This way, the monitor *does not need to handle the parameter information* and can employ any existing technique for ordinary, non-parametric traces, including state machines and pushdown automata, providing a formalism-independent way to check parametric properties. When an event comes, our algorithm will dispatch it to related monitors, which will update their states accordingly. For example, the seventh event in Figure 2, update_map$\langle m_1 \rangle$, will be dispatched to monitors for $\langle m_1, c_1 \rangle$, $\langle m_1, c_2 \rangle$, $\langle m_1, c_1, i_1 \rangle$, $\langle m_1, c_1, i_2 \rangle$, and $\langle m_1, c_2, i_3 \rangle$. New monitor instances will be created if the event contains new parameter instances. For example, when the third event in Figure 2, create_iter$\langle c_1, i_1 \rangle$, is received, a new monitor will be created for $\langle m_1, c_1, i_1 \rangle$ by combining $\langle m_1, c_1 \rangle$ in the first event with $\langle c_1, i_1 \rangle$. Detailed discussion about the monitoring algorithm can be found in Section 4.

An algorithm to build parameter instances from observed events, like the one introduced in (9), may create many useless monitor instances leading to prohibitive runtime overheads. For example, Figure 3 does not need to contain the binding $\langle m_1, c_3, i_4 \rangle$ even though it can be created by combining the parameter instances of update_map$\langle m_1 \rangle$ (the seventh event) and create_iter$\langle c_3, i_4 \rangle$ (the tenth event). It is safe to ignore this binding here because m_1 is not the underlying map for c_3, i_4. It is critical to minimize the number of monitor instances created during monitoring. The advantage is twofold: (1) that it reduces the needed memory space, and (2), more importantly, monitoring efficiency is improved since fewer monitors are triggered for each received event.

We present an effective solution in this paper to minimize the created monitors, based on the concept of the *enable set*, which is formally discussed in Section 5. An enable set is constructed for each event, say e, defined for a particular property. The enable set associated with e is a set of sets of parameters. Each of these sets of parameters denotes parameters that must have been seen before the arrival of event e, for e to be acceptable by a monitor instance. Consider the event update_map, it may occur anywhere in a matching trace, *except* for as the first event. Because the first event must be create_coll in a matching trace, and because create_coll instantiates both m and c, one of the sets in the enable set for update_map must be $\{m, c\}$. However, update_map may (in fact, must, to match the pattern) occur after the create_iter event. Because create_iter many not occur before create_coll we also have the set $\{m, c, i\}$ in the enable set for update_map. The final result for the enable set for update_map is thus: $\{\{m, c\}, \{m, c, i\}\}$. Therefore, when update_map$\langle m_1 \rangle$ arrives (the seventh event), the instance monitors for $\langle m_1, c_1 \rangle$ and $\langle m_1, c_2 \rangle$ must be updated because they bind $\{m, c\}$, and the instance monitors for $\langle m_1, c_1, i_1 \rangle$, $\langle m_1, c_1, i_2 \rangle$, and $\langle m_1, c_2, i_3 \rangle$ must be updated because they bind $\{m, c, i\}$, and have the same value for m (m_1). In this example all of the instances to update have already been created by the time the event arrives, but it should also be noted that no new instances can be created because at least m and c must be bound before update_map can occur.

It is worth mentioning that one may reduce the needed monitors using static program analysis, e.g., the one introduced in (5). However, such techniques are based on the program targeted for monitoring and lead to drawbacks in practice: (1) it is a more complex and thus slower analysis and (2) the analysis must be run for every target program, making the approach non-modular. For example, if the property to monitor is related to some library, one will have to run the analysis for every program using the library, which can be expensive, and often infeasible. The analysis needed by our approach, on the other hand, is usually much quicker[3], because properties tend to be much smaller than the programs they are designed to monitor. Moreover, our optimization technique requires no additional analysis when used in a situation, like for a library, where a property is checked for

[3] The analysis is upper bounded by the number of acyclic paths from the start state/symbol through a finite state machine/context free grammar, because convergence is achieved through one cycle. Finite state machines and context free grammars for properties tend to be small.

different programs, because the enable set is derived from the property itself instead of the targeted program.

3. Background: Parametric Monitoring

In this section, we briefly introduce the semantics of parametric monitoring based on parametric trace slicing. More details, including further formal definitions and proofs, can be found in (9). We include only the core definitions here to make this paper self-contained.

3.1 Events, Traces and Properties

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, ? 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.

DEFINITION 1. *Let $\mathcal{E}$ be a set of (non-parametric) events, called **base events** or simply **events**. An $\mathcal{E}$-**trace**, or simply a (non-parametric) **trace** when $\mathcal{E}$ is understood, 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 $e \in w$.*

For example, {create_coll, create_iter, use_iter, update_map} is the set of events from Figure 1, and create_coll create_iter use_iter update_map is a trace.

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

It is common, though not enforced, that $\mathcal{C}$ includes validating, violating, and don't know (or ?) categories, possibly with different names to capture the underlying intuition of the logic (e.g., match, like for ERE and CFG). For example, for the regular pattern in Figure 1, create_coll create_iter update_map use_iter is a matching trace, create_coll create_iter is a don't know trace if the trace is not finished, and create_coll update_map is a violating trace. In general, $\mathcal{C}$, the co-domain of P, can be any set (finite or infinite).

DEFINITION 3. *Let X be a set of **parameters** and let V be a set of corresponding **parameter values** (e.g., objects in Java). If $\mathcal{E}$ is a set of events events (Definition 1), then let $\mathcal{E}\langle X\rangle$ denote the set of corresponding **parametric events** $e\langle\theta\rangle$, where e is a base event in $\mathcal{E}$ and θ is a **parameter instance**, i.e., an element in $[X \overset{\circ}{\to} V]$, the set of partial maps from X to V. $\perp$ is the empty partial map. A **parametric trace** is a trace with events in $\mathcal{E}\langle X\rangle$, i.e., a word in $\mathcal{E}\langle X\rangle^*$.*

For example, if $X = \{\mathsf{m}, \mathsf{c}, \mathsf{i}\}$ is a set of parameters (of types {Map, Collection, Iterator}, respectively) and $V = \{m_1, c_1, i_1, i_2\}$, then create_coll$\langle \mathsf{m} \mapsto m_1, \mathsf{c} \mapsto c_1\rangle$, create_iter$\langle \mathsf{c} \mapsto c_1, \mathsf{i} \mapsto i_1\rangle$, and use_iter$\langle \mathsf{i} \mapsto i_1\rangle$, are parametric events and create_coll$\langle \mathsf{m} \mapsto m_1, \mathsf{c} \mapsto c_1\rangle$ create_iter$\langle \mathsf{c} \mapsto c_1, \mathsf{i} \mapsto i_1\rangle$ use_iter$\langle \mathsf{i} \mapsto i_1\rangle$ is a parametric trace. In practice, a parametric event usually instantiates a specific set of parameters, which are given in its *event definition*:

DEFINITION 4. *Let X be a set of parameters. If $\mathcal{E}$ is a set of base events like in Definition 1, we define a **parametric event definition**, or **event definition** for short, as a function $\mathcal{D}_\mathcal{E} : \mathcal{E} \to \mathcal{P}_f(X)$, where $\mathcal{P}_f$ is "finite power set", that maps an event to a set of parameters that will be instantiated by e at runtime. Parametric event $e\langle\theta\rangle$ is $\mathcal{D}_\mathcal{E}$-**consistent** if $Dom(\theta) = \mathcal{D}_\mathcal{E}(e)$. Parametric trace τ is $\mathcal{D}_\mathcal{E}$-**consistent** if $e\langle\theta\rangle$ is $\mathcal{D}_\mathcal{E}$-consistent for any $e\langle\theta\rangle \in \tau$.*

The example in Figure 1 contains the parametric event definition (create_coll $\mapsto$ {m, c}, create_iter $\mapsto$ {c, i}, use_iter $\mapsto$ {i}, update_map $\mapsto$ {m}). It states that two parameters, namely, m and c, will be instantiated at runtime when a parametric event create_coll$\langle\theta\rangle$ is received, etc. create_coll$\langle \mathsf{m} \mapsto m_1, \mathsf{c} \mapsto c_1\rangle$ is therefore one of its instances. All the parametric traces used in the remaining of this paper are assumed to follow certain given event definitions. Also, from here on we simplify the representation of parametric instances by hiding their domains when they are understood from the context. For example, given the above parametric event definition, we use create_coll$\langle m_1, c_1\rangle$ instead of create_coll$\langle \mathsf{m} \mapsto m_1, \mathsf{c} \mapsto c_1\rangle$, and $\langle m_1, c_1\rangle$ instead of $\langle \mathsf{m} \mapsto m_1, \mathsf{c} \mapsto c_1\rangle$.

DEFINITION 5. *Parameter instance θ is **compatible with** parameter instance θ' if for any parameter $x \in Dom(\theta) \cap Dom(\theta')$, $\theta(x) = \theta'(x)$. We can **combine** compatible parameter instances θ and θ', written $\theta \sqcup \theta'$, as follows:*

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

*θ' is **less informative** than θ, written $\theta' \sqsubseteq \theta$, if and only if for any $x \in X$, if $\theta'(x)$ is defined then $\theta(x)$ is also defined and $\theta'(x) = \theta(x)$. $\sqsubseteq$ is a partial order.*

With the notation above, $\langle m_1, c_1\rangle$ and $\langle c_1, i_1\rangle$ are compatible and $\langle m_1, c_1\rangle \sqcup \langle c_1, i_1\rangle = \langle m_1, c_1, i_1\rangle$. Logically, $\perp$ is compatible with, and less informative than, all parameter instances, because it does not bind any parameters.

DEFINITION 6. *Given parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$ and θ in $[X \overset{\circ}{\to} V]$, we let the θ-**trace slice** $\tau{\restriction_\theta} \in \mathcal{E}^*$ be the non-parametric trace in $\mathcal{E}^*$ defined as follows:*

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

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. Consider the parametric trace create_coll$\langle m_1, c_1\rangle$ create_iter$\langle c_1, i_1\rangle$ use_iter$\langle i_1\rangle$ update_map$\langle m_1\rangle$ create_coll $\langle m_1, c_2\rangle$. The trace slice for $\langle m_1\rangle$ is update_map, for $\langle m_1, c_1\rangle$ is create_coll update_map, for $\langle m_1, c_2\rangle$ is create_coll, and for $\langle m_1, c_1, i_1\rangle$ is create_coll create_iter use_iter update_map.

This definition of trace slicing is designed specifically for monitoring. Given a parametric property to monitor, the parameter set X used in the monitoring process is fixed accordingly. In other words, the monitoring process extracts from the observed execution a parametric trace containing only parameter bindings for parameters in X. Therefore, it is crucial to discard parameter instances that are not relevant to θ during the slicing, even including those more informative than θ, because, otherwise, monitors for incompatible parameter instances may interfere with one another, resulting in incorrect monitoring. For example, if a monitor is created for $\langle m_1, c_1\rangle$, it should not accept events containing information about i, e.g., create_iter$\langle c_1, i_1\rangle$ and create_iter$\langle c_1, i_2\rangle$. Otherwise, incompatible parameter instances $\langle m_1, c_1, i_1\rangle$ and $\langle m_1, c_1, i_2\rangle$ would "interfere" with each other in the parameter instance monitor for $\langle m_1, c_1\rangle$.

DEFINITION 7. *Let X be a set of parameters with their corresponding values V, like in Definition 3, and let $P : \mathcal{E}^* \to \mathcal{C}$ be a non-parametric property like in Definition 2. Then we define the* ***parametric property*** *$\Lambda X.P$ as the property (over traces $\mathcal{E}\langle X\rangle^*$ and categories $[[X \xrightarrow{\circ} V] \to \mathcal{C}]$)*

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

defined as $(\Lambda X.P)(\tau)(\theta) = P(\tau_{\uparrow\theta})$.

$\Lambda X.P$ 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.

It is worth noting that θ is a partial parameter binding and $\Lambda X.P$ is defined over traces that may not instantiate all the parameters in X. This makes it more expressive than the definition of parametric properties adopted by Tracematches (1; 2), which only supports parametric regular expressions such that *all* the parameters are instantiated whenever the pattern is matched by a trace. For example, consider a property where one wishes to match the start and possible use of a remote resource by a client. A regular expression to match this property would be start$\langle resource\rangle$ use$\langle client, resource\rangle^*$. The trace start$\langle resource_1\rangle$ should match the pattern, but it would not be matched in Tracematches, because there is no instantiation of the parameter $client$.

3.2 Parametric Monitors

We first define non-parametric monitors M as potentially infinite-state variants of Moore machines; then we define parametric monitors $\Lambda X.M$ as monitors maintaining one non-parametric monitor state per parameter instance.

DEFINITION 8. *A* ***monitor*** *M is a tuple $(S, \mathcal{E}, \mathcal{C}, 1, \sigma : S \times \mathcal{E} \to S, \gamma : S \to \mathcal{C})$, where S is a set of states, $\mathcal{E}$ is a set of input events, $\mathcal{C}$ is a set of output categories, $1 \in S$ 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 above notion of a monitor is rather conceptual. Actual implementations of monitors need not generate all the state space apriori, but on a "by need" basis. Allowing monitors with infinitely many states is a necessity. Even though only a finite number of states is reached during any given (finite) execution trace, there is, in general, no bound on the number of states. For example, monitors for context-free grammars like the ones in (16) have potentially unbounded stacks as part of their state. Also, as shown shortly, parametric monitors have domains of functions as state spaces, which are infinite as well. What is common to all monitors, though, is that they can take a trace event-by-event and, as each event is processed, classify the observed trace into a category. The following is natural:

DEFINITION 9. *$M = (S, \mathcal{E}, \mathcal{C}, 1, \sigma, \gamma)$ is a* ***monitor for property*** *$P : \mathcal{E}^* \to \mathcal{C}$ iff $\forall w \in \mathcal{E}^*, \gamma(\sigma(1, w)) = P(w)$.*

We next define parametric monitors: 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 10. *Given parameters X with corresponding values V and $M = (S, \mathcal{E}, \mathcal{C}, 1, \sigma : S \times \mathcal{E} \to S, \gamma : S \to \mathcal{C})$, we define the* ***parametric monitor*** *$\Lambda X.M$ as the monitor*

$$([[X \xrightarrow{\circ} V] \to S], \mathcal{E}\langle X\rangle, [[X \xrightarrow{\circ} V] \to \mathcal{C}], \lambda\theta.1, \Lambda X.\sigma, \Lambda X.\gamma),$$

with $\Lambda X.\sigma : [[X \xrightarrow{\circ} V] \to S] \times \mathcal{E}\langle X\rangle \to [[X \xrightarrow{\circ} V] \to S]$ and $\Lambda X.\gamma : [[X \xrightarrow{\circ} V] \to S] \to [[X \xrightarrow{\circ} V] \to \mathcal{C}]$

defined as $\forall \delta \in [[X \xrightarrow{\circ} V] \to S] \vee \forall \theta, \theta' \in [X \xrightarrow{\circ} V]$.

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

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

For the sake of rigor, Definition 10 may seem very complicated. All it says, however, is that 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). This is analogous to the intuitive example in Figure 3, where the non-parametric slice uniquely determines the state for the parameter instance, and

the Status column is the output category. Note that the ? in Status column stands for "don't know".

PROPOSITION 1. *If M is a monitor for property P then parametric monitor $\Lambda X.M$ is a monitor for parametric property $\Lambda X.P$. (See* (9)*)*

This means that we can construct a parametric monitor for a parametric property $\Lambda X.P$ by first creating a non-parametric base monitor for non-parametric property P, and then straight-forwardly extending it as per Definition 10.

In the next sections we discuss algorithms for efficient online monitoring of parametric properties $\Lambda X.P$, given a non-parametric monitor M for property P. We start with a base algorithm that extends algorithm $\mathbb{C}\langle X\rangle$ in (9) to support monitor *creation events*. Then we show that it can be significantly improved provided that enable sets for the property in question are available.

4. Monitoring with Creation Events : $\mathbb{C}^+\langle X\rangle$

As mentioned earlier, the monitor creation events are those events that are the first event in matching traces. The point of monitor creation events is to delay the expensive creation of monitor instances until a point where a pattern can actually be matched. For instance, using our example from Figure 1, there is no way a trace beginning with update_map$\langle m_1\rangle$ can ever match the patterns or validate the formulae, so creating a monitor instance for m_1 is a waste of both time and memory.

The first challenge to online monitoring of a parametric property is that the state space of potential parameter instances is infinite. Like in (9), we encode partial functions $[[X \overset{\circ}{\to} V] \overset{\circ}{\to} Y]$, which map some parameter instances $[X \overset{\circ}{\to} V]$ to elements in Y, as tables with entries indexed by parameter instances in $[X \overset{\circ}{\to} V]$ and with elements in Y. It can be easily seen that, in what follows, such tables will have a finite number of entries provided that each event instantiates a finite number of parameters, which is always the case.

Figure 5 shows the algorithm $\mathbb{C}^+\langle X\rangle$ for online monitoring of parametric property $\Lambda X.P$, given that M is a monitor for P. The algorithm shows which actions to perform, e.g., creating a new monitor state and/or updating the state of related monitors, when an event is received. It is a slightly different variant of algorithm $\mathbb{C}\langle X\rangle$ in (9). $\mathbb{C}^+\langle X\rangle$ is justified and motivated by experience with implementing and evaluating $\mathbb{C}\langle X\rangle$ in (9), mainly by the following observation: one often chooses to starting monitoring at the witness of a specific set of events (instead of monitoring from the beginning of the program). For example, when we monitor the property in Figure 1, we can choose to start monitoring on a pair of m and c objects, (m_1, c_1), only when a create_coll event is received, ignoring all the update_map$\langle m_1\rangle$ events before the creation. We call such events that lead to creation of new monitor states *(monitor) creation events*. Algorithm $\mathbb{C}^+\langle X\rangle$ extends $\mathbb{C}\langle X\rangle$ in (9) to support creation events. It is easy to see that $\mathbb{C}\langle X\rangle$ can be regarded as a special case of

```
Algorithm ℂ⁺⟨X⟩(M = (S, ℰ, 𝒞, ι, σ, γ))
Globals: mapping Δ : [[X ⇢° V] ⇢° S]
         mapping 𝒰 : [X ⇢° V] → 𝒫_f([X ⇢° V])
Initialization: 𝒰(θ) ← ∅ for any θ ∈ [X ⇢° V]

function main(e⟨θ⟩)
 1 if Δ(θ) undefined then
 2 ⋮ foreach θ_m ⊏ θ (in reversed topological order) do
 3 ⋮ ⋮ if Δ(θ_m) defined then
 4 ⋮ ⋮ ⋮ goto 7
 5 ⋮ ⋮ endif
 6 ⋮ endfor
 7 ⋮ if Δ(θ_m) defined then
 8 ⋮ ⋮ defineTo(θ, θ_m)
 9 ⋮ elseif e is a creation event then
10 ⋮ ⋮ defineNew(θ)
11 ⋮ endif
12 ⋮ foreach θ_m ⊏ θ (in reversed topological order) do
13 ⋮ ⋮ foreach θ_comp ∈ 𝒰(θ_m) compatible with θ do
14 ⋮ ⋮ ⋮ if Δ(θ_comp ⊔ θ) undefined then
15 ⋮ ⋮ ⋮ ⋮ defineTo(θ_comp ⊔ θ, θ_comp)
16 ⋮ ⋮ ⋮ endif
17 ⋮ ⋮ endfor
18 ⋮ endfor
19 endif
20 foreach θ' ∈ {θ} ∪ 𝒰(θ) do
21 ⋮ Δ(θ') ← σ(Δ(θ'), e)
22 endfor
function defineNew(θ)
1 Δ(θ) ← ι
2 foreach θ'' ⊏ θ do
3 ⋮ 𝒰(θ'') ← 𝒰(θ'') ∪ {θ}
4 endfor
function defineTo(θ, θ')
1 Δ(θ) ← Δ(θ')
2 foreach θ'' ⊏ θ do
3 ⋮ 𝒰(θ'') ← 𝒰(θ'') ∪ {θ}
4 endfor
```

Figure 5. Monitoring Algorithm $\mathbb{C}^+\langle X\rangle$.

$\mathbb{C}^+\langle X\rangle$, when all the events are creation events. Note that (9) used creation events in the evaluation, but they were not formalized in the algorithm. The proof of $\mathbb{C}^+\langle X\rangle$ is tedious, but is easily derived from the proof of $\mathbb{C}\langle X\rangle$ in (9).

Two mappings are used: Δ and $\mathcal{U}$. Δ stores the monitor states for parameter instances , and $\mathcal{U}$ maps a parameter instance θ to *all the parameter instances* that have been defined and are properly more informative than θ. In what follows, "the monitor state for θ" refers to $\Delta(\theta)$ to facilitate reading in some contexts, and, accordingly, "to create a parameter instance θ" and "to create a monitor state for parameter instance θ" have the same meaning: to define $\Delta(\theta)$.

Event	update_map$\langle m_1 \rangle$	create_coll$\langle m_1, c_1 \rangle$	create_coll$\langle m_2, c_2 \rangle$	create_iter$\langle c_1, i_1 \rangle$
Δ	$\emptyset$	$\langle m_1, c_1 \rangle$: $\sigma(i,$ create_coll$)$	$\langle m_1, c_1 \rangle$: $\sigma(i,$ create_coll$)$ $\langle m_2, c_2 \rangle$: $\sigma(i,$ create_coll$)$	$\langle m_1, c_1 \rangle$: $\sigma(i,$ create_coll$)$ $\langle m_2, c_2 \rangle$: $\sigma(i,$ create_coll$)$ $\langle m_1, c_1, i_1 \rangle$: $\sigma(\sigma(i,$ create_coll$),$ create_iter$)$
$\mathcal{U}$	$\emptyset$	$\bot : \langle m_1, c_1 \rangle$ $\langle m_1 \rangle$: $\langle m_1, c_1 \rangle$ $\langle c_1 \rangle$: $\langle m_1, c_1 \rangle$	$\bot : \langle m_1, c_1 \rangle, \langle m_2, c_2 \rangle$ $\langle m_1 \rangle$: $\langle m_1, c_1 \rangle$ $\langle c_1 \rangle$: $\langle m_1, c_1 \rangle$ $\langle m_2 \rangle$: $\langle m_2, c_2 \rangle$ $\langle c_2 \rangle$: $\langle m_2, c_2 \rangle$	$\bot : \langle m_1, c_1 \rangle, \langle m_2, c_2 \rangle, \langle m_1, c_1, i_1 \rangle$ $\langle m_1 \rangle$: $\langle m_1, c_1 \rangle, \langle m_1, c_1, i_1 \rangle$ $\langle c_1 \rangle$: $\langle m_1, c_1 \rangle, \langle m_1, c_1, i_1 \rangle$ $\langle m_2 \rangle$: $\langle m_2, c_2 \rangle$ $\langle c_2 \rangle$: $\langle m_2, c_2 \rangle$ $\langle i_1 \rangle$: $\langle m_2, c_2 \rangle, \langle m_2, c_2, i_1 \rangle$ $\langle m_1, c_1 \rangle$: $\langle m_1, c_1, i_1 \rangle$ $\langle m_1, i_1 \rangle$: $\langle m_1, c_1, i_1 \rangle$ $\langle c_1, i_1 \rangle$: $\langle m_1, c_1, i_1 \rangle$

Figure 6. Sample Run of $\mathbb{C}^+\langle X \rangle$. The first row gives the received events; the second and the third rows give the content of Δ and $\mathcal{U}$, respectively, after every event is processed. Monitor states are represented symbolically in the table, e.g., $\sigma(i,$ create_coll$)$ represents the state after a monitor processes event create_coll.

When parametric event $e\langle\theta\rangle$ arrives, the algorithm first checks whether θ has been encountered yet by checking if its corresponding monitor state, i.e., $\Delta(\theta)$, has been defined(line 1 in main). If θ is encountered for the first time, new parameter instances may need be created. In such a case, we first try to locate the maximum parameter instance (θ_m) which is less informative than θ and for which a monitor state has been created (lines 2 - 6). If such θ_m is found, its monitor state is used to initialize the monitor state for θ (lines 7 and 8); otherwise, a new monitor state is created for θ *only if* e is a creation event (lines 9 and 10). Also, new parameter instances can be created by combining θ with existing parameter instances that are compatible with θ, i.e., they do not have conflicting parameter bindings. An observation here is that if parameter instance θ_{comp} has been created and is compatible with θ then θ_{comp} can be found in $\mathcal{U}(\theta_m)$ for some $\theta_m \sqsubset \theta$ according to the definition of $\mathcal{U}$. Therefore, algorithm $\mathbb{C}^+\langle X \rangle$ searches through all the $\theta_m \sqsubset \theta$ to find all possible θ_{comp}, examining whether any new parameter instance should be created (lines 12 - 17).

If θ has been seen before, or otherwise after all the new monitor states have been created/initialized as explained above, algorithm $\mathbb{C}^+\langle X \rangle$ invokes all the monitors that need to process e, namely, those whose corresponding parameter instances are more informative than or equal to θ (lines 20 - 22). The updates make use of the sets stored in $\mathcal{U}$ to know which instances are more informative (line 20). There are two auxiliary functions: defineNew and defineTo. The former initializes a new monitor state for the input parameter instance and the latter creates a monitor state for the first input parameter instance using the monitor state for the second instance. Both functions add θ to the sets in table $\mathcal{U}$ for the bindings less informative than θ.

We next use an example run, illustrated in Figure 6, to show how $\mathbb{C}^+\langle X \rangle$ works. In Figure 6, we show the contents of Δ and $\mathcal{U}$ after every event (given in the first row of the table) is processed. The observed trace is update_map$\langle m_1 \rangle$ create_coll$\langle m_1, c_1 \rangle$ create_coll $\langle m_2, c_2 \rangle$ create_iter$\langle c_1, i_1 \rangle$. We assume that create_coll is the only creation event.

The first event, update_map$\langle m_1 \rangle$, is not a creation event and nothing is added to Δ and $\mathcal{U}$. The second event, create_coll$\langle m_1, c_1 \rangle$, is a creation event. So a new monitor state is defined in Δ for $\langle m_1, c_1 \rangle$, which is also added to the lists in $\mathcal{U}$ for $\bot$, $\langle m_1 \rangle$ and $\langle c_1 \rangle$. Note that $\bot$ is less informative than any other parameter instances. The third event create_coll$\langle m_2, c_2 \rangle$ is another creation event, incompatible with the second event . Hence, only one new monitor state is added to Δ. $\mathcal{U}$ is updated similarly. The last event create_iter$\langle c_1, i_1 \rangle$ is not a creation event. So no monitor instance is created for $\langle c_1, i_1 \rangle$. It is compatible with the existing parameter instance $\langle m_1, c_1 \rangle$ introduced by the second event but not compatible with $\langle m_2, c_2 \rangle$ due to the conflict binding on c. The compatible instance $\langle m_1, c_1 \rangle$ can be found from the list for $\langle c_1 \rangle$ in $\mathcal{U}$. Therefore, a new monitor instance is created for the combined parameter instance $\langle m_1, c_1, i_1 \rangle$ using the state for $\langle m_1, c_1 \rangle$ in Δ. $\mathcal{U}$ is also updated to add the combined parameter instance into lists of parameter instances that are less informative.

5. Limitations of $\mathbb{C}^+\langle X \rangle$ and Enable Sets

$\mathbb{C}^+\langle X \rangle$ does not make any assumption on the given monitor M. In other words, one may monitor properties written in any specification formalism, e.g., ERE, CFG, PTLTL etc., as long as one also provides a monitor generation algorithm for said formalism. However, this generality leads to extra monitoring overhead in some cases. Thus we introduce our novel optimization based on the concept of enable sets.

To motivate the optimization, let us continue the run in Figure 6 to process one more event, use_iter$\langle i_1 \rangle$. The result is shown in Figure 7. use_iter$\langle i_1 \rangle$ is not a creation event and no monitor instance is created for $\langle i_1 \rangle$. Since $\langle i_1 \rangle$ is compatible with $\langle m_2, c_2 \rangle$, a new monitor instance is defined for $\langle m_2, c_2, i_1 \rangle$. The monitor instance for $\langle m_1, c_1, i_1 \rangle$ is then updated according to use_iter because $\langle i_1 \rangle$ is less informative than $\langle m_1, c_1, i_1 \rangle$. $\mathcal{U}$ is also updated to add $\langle m_2, c_2, i_1 \rangle$ to

the lists for all the parameter instances less informative than $\langle m_2, c_2, i_1\rangle$. New entries are added into $\mathcal{U}$ during the update since some of less informative parameter instances, e.g., $\langle m_2, i_1\rangle$, have not been used before this event.

Event	use_iter$\langle i_1\rangle$
Δ	$\langle m_1, c_1\rangle$: $\sigma(i, \text{create\_coll})$ $\langle m_2, c_2\rangle$: $\sigma(i, \text{create\_coll})$ $\langle m_1, c_1, i_1\rangle$: $\sigma(\sigma(\sigma(i, \text{create\_coll}), \text{create\_iter}), \text{use\_iter})$ $\langle m_2, c_2, i_1\rangle$: $\sigma(\sigma(i, \text{create\_coll}), \text{use\_iter})$
$\mathcal{U}$	$\bot : \langle m_1, c_1\rangle, \langle m_2, c_2\rangle, \langle m_2, c_2, i_1\rangle, \langle m_1, c_1, i_1\rangle$ $\langle m_1\rangle$: $\langle m_1, c_1\rangle, \langle m_1, c_1, i_1\rangle$ $\langle c_1\rangle$: $\langle m_1, c_1\rangle, \langle m_1, c_1, i_1\rangle$ $\langle m_2\rangle$: $\langle m_2, c_2\rangle, \langle m_2, c_2, i_1\rangle$ $\langle c_2\rangle$: $\langle m_2, c_2\rangle, \langle m_2, c_2, i_1\rangle$ $\langle i_1\rangle$: $\langle m_2, c_2, i_1\rangle, \langle m_1, c_1, i_1\rangle$ $\langle m_2, \mathsf{c} \mapsto c_2\rangle$: $\langle m_2, c_2, i_1\rangle$ $\langle m_2, i_1\rangle$: $\langle m_2, c_2, i_1\rangle$ $\langle c_2, i_1\rangle$: $\langle m_2, c_2, i_1\rangle$ $\langle m_1, c_1\rangle$: $\langle m_1, c_1, i_1\rangle$ $\langle m_1, i_1\rangle$: $\langle m_1, c_1, i_1\rangle$ $\langle c_1, i_1\rangle$: $\langle m_1, c_1, i_1\rangle$

Figure 7. Following the Run of Figure 6.

Creating the monitor instance for $\langle m_2, c_2, i_1\rangle$ is needed for the correctness of $\mathbb{C}^+\langle X\rangle$, but it can be avoided when more information about the program or the specification is available. For example, according to the semantics of Iterator, no event create_iter$\langle c_2, i_1\rangle$ will occur in the following execution since an iterator can be associated to only one collection. Hence, the monitor for $\langle m_2, c_2, i_1\rangle$ will never reach the validation state and we do not need to create it from the beginning. However, such semantic information about the program is very difficult to infer automatically. Below, we show a simpler yet effective solution to avoid unnecessary monitor creations by analyzing the specification to monitor.

When monitoring a program against a specific property, usually only a certain subset of property categories, ($\mathcal{C}$ in Definition 2), is checked. For example, in Figure 1, the regular expression specifies a defective interaction among related Map, Collection and Iterator objects. To find an error in the program using monitoring is thus to detect matches of the specified pattern during the execution. In other words, we are only interested in the validation category of the specified pattern. Obviously, to match the pattern, for a parameter instance of parameter set {m, c, i}, create_coll and create_iter should be observed before use_iter is encountered for the first time in monitoring. Otherwise, the trace slice for {m, c, i} will never match the pattern. Based on this information, we next show that creating the monitor state for $\langle m_2, c_2, i_1\rangle$ in Figure 7 is not needed. When event use_iter$\langle i_1\rangle$ is encountered, if the monitor state for a parameter instance $\langle m_2, c_2\rangle$ exists without the monitor state for $\langle m_2, c_2, i_1\rangle$, like in Figure 7, it can be inferred that in the trace slice for $\langle m_2, c_2, i_1\rangle$, only events create_coll and/or update_map occur before use_iter because, otherwise, if create_iter also occurred before use_iter, the monitor state for $\langle m_2, c_2, i_1\rangle$ should have been created. Therefore, we can infer, when event use_iter$\langle i_1\rangle$

Event	$\text{enable}^{\mathcal{E}}_{\mathcal{G}}$(Event)
create_coll	$\{\emptyset\}$
create_iter	$\{\{\text{create\_coll}\},$ $\{\text{create\_coll, update\_map}\}\}$
use_iter	$\{\{\text{create\_coll, create\_iter}\},$ $\{\text{create\_coll, create\_iter, update\_map}\}\}$
update_map	$\{\{\text{create\_coll}\},$ $\{\text{create\_coll, create\_iter}\},$ $\{\text{create\_coll, create\_iter, use\_iter}\}\}$

Figure 8. Property Enable Set for UnsafeMapIterator.

is observed and before the execution continues, that no match of the specified pattern can be reached by the trace slice for $\langle m_2, c_2, i_1\rangle$, that is to say, the monitor for $\langle m_2, c_2, i_1\rangle$ will never reach the validation state.

This observation shows that the knowledge about the specified property can be applied to avoid unnecessary creation of monitor states. This way, the sizes of Δ and $\mathcal{U}$ can be reduced, reducing the monitoring overhead. We next formalize the information needed for the optimization and argue that it is not specific to the underlying specification formalism, and that it can be computed easily. How this information is used is discussed in Section 6.

5.1 Enable Sets

DEFINITION 11. *Given $\tau \in \mathcal{E}^*$ and $e, e' \in \tau$, we denote that e' occurs before the first occurrence of e in τ as $e' \rightsquigarrow_\tau e$. Let the **trace enable set** of $e \in \mathcal{E}$ be the function $\text{enable}_\tau : \mathcal{E} \to \mathcal{P}_f(\mathcal{E})$, defined as: $\text{enable}_\tau(e) = \{e' \mid e' \rightsquigarrow_\tau e\}$.*

Note that if $e \notin \tau$ then $\text{enable}_\tau(e) = \emptyset$. The trace enable set can be used to examine whether the execution under observation may generate a particular trace of interest, or not: if event e is encountered during monitoring but some event $e' \in \text{enable}_\tau(e)$ has not been observed, then the (incomplete) execution being monitored will *not* produce the trace τ when it finishes. This observation can be extended to check, before an execution finishes, whether the execution can generate a trace belonging to some designated property categories. The designated property categories are called the *goal* of the monitoring in what follows.

DEFINITION 12. *Given $P : \mathcal{E}^* \to \mathcal{C}$ and a set of categories $\mathcal{G} \subseteq \mathcal{C}$ as the goal, the **property enable set** is defined as a function $\text{enable}^{\mathcal{E}}_{\mathcal{G}} : \mathcal{E} \to \mathcal{P}_f(\mathcal{P}_f(\mathcal{E}))$ with $\text{enable}^{\mathcal{E}}_{\mathcal{G}}(e) = \{\text{enable}_\tau(e) \mid P(\tau) \in \mathcal{G}\}$.*

Intuitively, if event e is encountered during monitoring but none of event sets $\text{enable}^{\mathcal{E}}_{\mathcal{G}}(e)$ has been completely observed, the (incomplete) execution being monitoring will not produce a trace τ s.t. $P(\tau) \in \mathcal{G}$. For example, given the property specified by the ERE-based property in Figure 1, where $\mathcal{G}$ contains only the match, violation, and ? categories, Figure 8 shows the property enable set for UnsafeMapIterator.

The property enable set provides a sound and fast way to decide whether an incomplete trace slice has the possibility of reaching the desired categories by looking at the events that have already occurred. In the above example, if a trace slice starts with create_coll use_iter, it will never reach the match category, because $\{\text{create\_coll}\} \notin \text{enable}^{\mathcal{E}}_{\mathcal{G}}(\text{use\_iter})$. In such case, no monitor state need be created even when the newly observed event may lead to new parameter instances. For example, suppose that the observed (incomplete) trace is $\text{create\_coll}\langle m_1, c_1\rangle$ $\text{use\_iter}\langle i_1\rangle$. At the second event, $\text{use\_iter}\langle i_1\rangle$, a new parameter instance can be constructed, namely, $\langle m_1, c_1, i_1\rangle$, and a monitor state s will be created for $\langle m_1, c_1, i_1\rangle$ if algorithm $\mathbb{C}^+\langle X\rangle$ is applied. However, since the trace slice for s is create_coll use_iter, we can immediately know that s cannot reach the match state, and thus there is no need to create and maintain s during monitoring if match is the target category.

A direct application of the above idea to optimize $\mathbb{C}^+\langle X\rangle$ requires maintaining observed events for every created monitor and comparing event sets when a new parameter instance is found, reducing the improvement of performance. Therefore, we extend the notion of the enable set to be based on parameter sets instead of event sets.

DEFINITION 13. *Given a property $P : \mathcal{E}^* \to \mathcal{C}$, a set of categories $\mathcal{G} \subseteq \mathcal{C}$ as the goal, a set of parameters X and a parameter definition $\mathcal{D}_\mathcal{E}$, the* ***property parameter enable set*** *of event $e \in \mathcal{E}$ is defined as a function $\text{enable}^X_\mathcal{G} : \mathcal{E} \to \mathcal{P}_f(\mathcal{P}_f(X))$ as follows: $\text{enable}^X_\mathcal{G}(e) = \{\cup\{\mathcal{D}_\mathcal{E}(e') \mid e' \in \text{enable}_\tau(e)\} \mid P(\tau) \in \mathcal{G}\}$.*

From now on, we use "enable set" to refer to "property parameter enable set" for simplicity. For example, given the ERE-based property in Figure 1 and $\mathcal{G} = \{\text{validating}\}$; Figure 9 shows the parameter enable set for UnsafeMapIterator. Then, given again the trace $\{\text{create\_coll}\}\langle m_1, c_1\rangle$ $\text{use\_iter}\langle i_1\rangle$, no monitor state need be created at the second event for $\langle m_1, c_1, i_1\rangle$ since the parameter instance used to initialize the new monitor state, namely, $\langle m_1, c_1\rangle$, is not in $\text{enable}^X_\mathcal{G}(\text{use\_iter})$. In other words, one may simply compare the parameter instance used to initialize the new parameter instance with the enable set of the observed event to decide whether a new monitor state is needed or not. Note that in JavaMOP, the property parameter enable sets are generated from the property enable sets provided by the formalism plugin in question. This allows the plugins to remain totally parameter agnostic. The following result guarantees the correctness of this approach:

PROPOSITION 2. *When algorithm $\mathbb{C}^+\langle X\rangle$ receives event $e\langle\theta\rangle$, if we use θ' to define $\theta \sqcup \theta'$ and $\text{Dom}(\theta') \notin \text{enable}^X_\mathcal{G}(e)$, then $\Delta(\theta \sqcup \theta') \notin \mathcal{G}$ during the whole monitoring process.*

5.2 Computing Enable Sets

The definition of the enable set is general and does not depend on a specific formalism to write the property. Although computing the enable set from a specified property requires understanding of the used formalism. It can be achieved as a "side-effect" of the monitor generation process, in which full knowledge about the property is available.

Event	$\text{enable}^X_\mathcal{G}$(Event)
create_coll	$\{\emptyset\}$
create_iter	$\{\{\text{m, c}\}\}$
use_iter	$\{\{\text{m, c, i}\}\}$
update_map	$\{\{\text{m, c}\}, \{\text{m, c, i}\}\}$

Figure 9. Parameter Enable Set for UnsafeMapIterator.

```
Algorithm EN_fsm(FSM = (E, S, s_0, δ, F))
Globals: mapping V_μ : S → P_f(P_f(E))
         mapping enable^E_G : E → P_f(P_f(E))
         set R ⊆ S

Initialization: fix G' ⊆ S, compute R for G'

function main()
 1 auxiliary(s_0, ∅)
function auxiliary(s, μ)
 1 foreach e ∈ E do
 2 ⋮ if δ(s, e) ∈ R then
 3 ⋮ ⋮ enable^E_G(e) ← enable^E_G(e) ∪ {μ − e}
 4 ⋮ endif
 5 ⋮ let μ' ← μ ∪ {e}
 6 ⋮ ⋮ if μ' ∉ V_μ(s)
 7 ⋮ ⋮ ⋮ V_μ(s) ← V_μ(s) ∪ {μ'}
 8 ⋮ ⋮ ⋮ auxiliary(δ(s, e), μ')
 9 ⋮ ⋮ endif
10 endfor
```

Figure 10. FSM g $\text{enable}^\mathcal{E}_\mathcal{G}$ Computation Algorithm.

Case 1: FSM The algorithm in Figure 10 computes the property enable sets for a finite state machine. We use this algorithm to compute the enable sets for any logic that is reducible to a finite state machine, including ERE, PTLTL, and FTLTL. The algorithm assumes a finite state machine, defined as $FSM = (\mathcal{E}, S, s_0 \in S, \delta : S \times \mathcal{E} \to S, F \subseteq S)$. $\mathcal{E}$ is the alphabet, traditionally listed as Σ but changed for consistency, since the alphabets of our FSMs are event sets. s_0 is the start state, corresponding to 1 in the definition of a monitor. δ is the transition function, taking a state and an event and mapping to a next state for the machine. F is the set of accept states. In the initialization we compute goal reachability set S by fixing a goal G' as an arbitrary set of states, such as the error state for violation, or accept states for matching a pattern originally specified as an ERE. More specifically, G' is the subset of S corresponding to the subset of $\mathcal{G}$ in which we are interested. For state $s \in S$, $s \in R$ if and only if there is a path from s to an $s' \in G'$. It is computed using a straight-forward depth first search from the initial state. $\mathcal{V}_\mu$ is a mapping from states to sets of

events; it is used to check for algorithm termination. $\mathsf{enable}_{\mathcal{G}}^{\mathcal{E}}$ is the output property enable set, which is converted into a parameter enable set by JavaMOP.

Function auxiliary is first called with $\mu = \emptyset$ and the initial state s_0 (the Initialization section). If we think of the FSM as a graph, μ represents the set of edges we have seen at least once in a traversal. For each event in $\mathcal{E}$ (line 1), we check to see if the next state, computed by $\delta(s, e)$ reaches our goal (line 2). If it does, that means we have seen a viable prefix set. From the definition of $\mathsf{enable}_{\mathcal{G}}^{\mathcal{E}}$, we know we need to add this prefix set to $\mathsf{enable}_{\mathcal{G}}^{\mathcal{E}}$ for the event e, which we do (line 3). Also on line 3, we make sure that we remove e from μ, as an enable set for e is not supposed to contain e. Line 5 begins the recursive step of the algorithm. We let $\mu' = \mu \cup \{e\}$, because we have traversed another edge, and that edge is labeled as e. The map $\mathcal{V}_\mu$ tells us which μ have been seen in previous recursive steps, in a given state. If a μ has been seen before, in a state, taking a recursive step can add no new information. Because of this, line 6 ensures that we only call the recursive step on line 8, if new information can be added. Line 7 keeps $\mathcal{V}$ consistent. Thus the algorithm terminates only when every viable μ has been seen in every reachable state, effectively computing a fixed point.

Case 2: CFG We also provide an algorithm to compute the enable set for a context-free pattern, which has an infinite monitor state space, as briefly explained in what follows[4]. This is a modification of the algorithm in Figure 10.

Let $\mathcal{G} = \{\mathsf{match}\}$. For $\mathsf{enable}_{\mathcal{G}}^{\mathcal{E}}$ and a given CFG $G = (NT, \mathcal{E}, P, S)$ we begin with all productions $S \to \gamma$ and the set $\mu_0 = \emptyset \in \mathcal{P}_f(\mathcal{E})$. For each production, we investigate each $s \in \gamma$ (where $\in$ is, by abuse of notation, used to denote a symbol in a right hand side) from left to right. If $s \in \mathcal{E}$ we add μ_i to $\mathsf{enable}_{\mathcal{G}}^{\mathcal{E}}(s)$, thus if s is the first symbol in γ we add μ_0. We then add s to μ_i forming μ_{i+1}. If $s \in NT$ we recursively invoke the algorithm, but rather than use μ_0, we use μ_i, and each production investigated will be of the form $s \to \gamma$. We keep track of which $s \in NT$ have been processed, to ensure termination.

Discussion. The general definition of the enable set allows us to separate the concerns of generating efficient monitoring code. On the framework level, such as the algorithms discussed in this paper, we can focus on applying the information encoded in the enable set to generate an efficient monitoring process for parametric properties, while on the logic level, where a monitor is generated for a given non-parametric property written in a specific formalism, one can focus on creating the fastest monitor that verifies the input trace against the property and also on producing the enable set information. The enable set represents static information about the given property and only need be generated once. As mentioned, the static analysis presented in (5), while effective, requires a complex analysis of the target program, which must be performed for every program one wants to monitor.

Other possibilities for optimization are exhibited in the example in Figure 7. We discuss two of them here. The first is to make use of the semantics of the program. In this example, we know that an i object is created from a c object and does not relate to other c objects. Hence, we can avoid creating a combination of $\langle m_2, c_2\rangle$ and $\langle i_1\rangle$ because i_1 is created from c_1. However, such semantic information is very difficult to achieve automatically and may require human input. The enable set, on the contrary, can be easily computed by statically analyzing the specification without analyzing any program or human interferences; indeed, the specified property already indicates some semantics of the involved parameters. Nevertheless, we believe that static analysis on the program to monitor, such as that in (5), can and should be applied in conjunction with enable sets to further reduce the monitoring overhead, whenever it is feasible.

Other optimizations are based on heuristics. One reasonable heuristic which can be applied here is that we may only combine parameter instances that are connected to one another through some events which have been observed (we cannot rely on future events in online monitoring). For example, $\langle i_1\rangle$ and $\langle m_1, c_1\rangle$ need to be combined to build a new parameter instance because c_1 and i_1 are connected in the second event, $\mathsf{create\_coll}\langle m_1, c_1\rangle$, in Figure 7, but $\langle i_1\rangle$ and $\langle m_2, c_2\rangle$ should not combined due to the heuristic. The intuition is that if two parameter instances do not interact in any event, it may imply that they are not relevant to each other even if they are compatible. However, because no information about future events available, such a heuristic can break, for example, an event connecting the two parameter instances comes afterward. The enable set provides a sound optimization, and we believe that it performs as well as, if not better than, such heuristics in most cases.

6. Monitoring with Enable Sets: $\mathbb{D}\langle X\rangle$

In this section we integrate the concept of enable sets with algorithm $\mathbb{C}^+\langle X\rangle$, to improve performance and memory usage. To ease reading, all proofs related to this algorithm can be found in Section 6.2.

Given a set of desired value categories $\mathcal{G}$, Proposition 2 guarantees that we can omit creating monitor states for certain parameter instances when an event is received using the enable set without missing any trace belonging to $\mathcal{G}$. However, skipping the creation of monitor states may result in false alarms, i.e., a trace that is not in $\mathcal{G}$ can be reported to belong to $\mathcal{G}$. Let us consider the following example. We monitor to find matching of a regular pattern e_1e_3 and the event definition is ($e_1 \mapsto \{P_1\}, e_2 \mapsto \{P_2\}, e_3 \mapsto \{P_1, P_2\}$) the observed trace is $e_1\langle p_1\rangle e_2\langle p_2\rangle e_3\langle p_1, p_2\rangle$. Also, suppose e_1 is the only creation event. Obviously, the trace does not match the pattern. Figure 11 shows the run using the optimization based on the enable set. Only the content of Δ is given for simplicity. At e_1, a

[4] We assume a certain familiarity with context free patterns; definitions can be found in (16), together with explanations on CFG monitoring.

monitor state is created for $\langle p_1 \rangle$ since it is the creation event. At e_2, no action is taken since $\mathsf{enable}^X_{\mathcal{G}}(e_2) = \emptyset$. At e_3, a monitor state will be created for $\langle p_1, p_2 \rangle$ using the monitor state for $\langle P_1 \mapsto p_1 \rangle$ since $\mathsf{enable}^X_{\mathcal{G}}\, e_3 = \{P_1\}$. This way, e_2 is forgotten and a match of the pattern is reported even though it is not correct to do so.

Event	$e_1\langle p_1 \rangle$	$e_2\langle p_2 \rangle$	$e_3\langle p_1, p_2 \rangle$
Δ	$\langle p_1 \rangle : \sigma(i, e_1)$	$\langle p_1 \rangle : \sigma(i, e_1)$	$\langle p_1 \rangle : \sigma(i, e_1)$ $\langle p_1, p_2 \rangle : \sigma(\sigma(i, e_1), e_3)$

Figure 11. Unsound Usage of the Enable Set.

6.1 Timestamping Monitors: Algorithm $\mathbb{D}\langle X \rangle$

To avoid unsoundness, we introduce the notion of disable stamps of events. $\mathsf{disable} : [[X \overset{\circ}{\to} V] \overset{\circ}{\to} \mathsf{integer}]$ maps a parameter instance to an integer timestamp. $\mathsf{disable}(\theta)$ gives the time when the last event with θ was received. We maintain timestamps for monitors using a mapping $\mathcal{T} : [[X \overset{\circ}{\to} V] \overset{\circ}{\to} \mathsf{integer}]$. $\mathcal{T}$ maps a parameter instance for which a monitor state is defined to the time when the original monitor state is created from a creation event. Specifically, if a monitor state for θ is created using the initial state when a creation event is received (i.e., using the defineNew function in algorithm $\mathbb{C}^+\langle X \rangle$), $\mathcal{T}(\theta)$ is set to the time of creation; if a monitor state for θ is created from the monitor state for θ', $\mathcal{T}(\theta')$ is passed to $\mathcal{T}(\theta)$. Figure 12 shows the evolution of disable and $\mathcal{T}$ while processing the trace in Figure 11.

disable and $\mathcal{T}$ can be used together to track "skipped events": when a monitor state for θ is created using the monitor state for θ', if there exists some $\theta'' \sqsubset \theta$ s.t. $\theta'' \not\sqsubseteq \theta'$ and $\mathsf{disable}(\theta'') > \mathcal{T}(\theta')$ then the trace slice for θ does not belong to the desired value categories $\mathcal{G}$. Intuitively, $\mathsf{disable}(\theta'') > \mathcal{T}(\theta')$ implies that an event $e\langle\theta''\rangle$ has been encountered after the monitor state for θ' was created. But θ'' was not taken into account ($\theta'' \not\sqsubseteq \theta'$). The only possibility is that e is omitted due to the enable set and thus the trace slice for θ does not belong to $\mathcal{G}$ according to the definition of the enable set. Therefore, in Figure 12, no monitor instance is created for $\langle p_1, p_2 \rangle$ at e_3 because $\mathsf{disable}(\langle p_2 \rangle) > \mathcal{T}(\langle p_1 \rangle)$.

Event	$e_1\langle p_1 \rangle$	$e_2\langle p_2 \rangle$	$e_3\langle p_1, p_2 \rangle$
Δ	$\langle p_1 \rangle : \sigma(i, e_1)$	$\langle p_1 \rangle : \sigma(i, e_1)$	$\langle p_1 \rangle : \sigma(i, e_1)$
$\mathcal{T}$	$\langle p_1 \rangle : 1$	$\langle p_1 \rangle : 1$	$\langle p_1 \rangle : 1$
disable	$\langle p_1 \rangle : 2$	$\langle p_1 \rangle : 2$ $\langle p_2 \rangle : 3$	$\langle p_1 \rangle : 2$ $\langle p_2 \rangle : 3$ $\langle p_1, p_2 \rangle : 4$

Figure 12. Sound Monitoring Using Enable Sets and Timestamps.

The above discussion applies when the skipped event occurs after the initial creation of the monitor state. The other case, i.e., an event is omitted before the initial monitor state is created, can also be handled using timestamps. First, if the skipped event is not a creation event, it does not affect the soundness of the algorithm to omit the event because of the definition of creation events. In the above example, if the observed trace is $e_2\langle p_2 \rangle e_1\langle p_1 \rangle e_3\langle p_1, p_2 \rangle$, we will ignore e_2 and report the matching at e_3 since e_1 is the only creation event. The situation becomes more sophisticated when the skipped event is a creation event. For example, we assume that both e_1 and e_2 are creation events in the above example. Figure 13 then shows the monitoring process for the parametric trace $e_2\langle p_2 \rangle e_1\langle p_1 \rangle e_3\langle p_1, p_2 \rangle$.

At e_2, $\Delta(\langle p_2 \rangle)$ is defined because it is a creation event. At e_1, $\Delta(\langle p_1 \rangle)$ is defined, but no monitor state is created for $\langle p_1, p_2 \rangle$ because $\{P_2\} \notin \mathsf{enable}^X_{\mathcal{G}}(e_1)$. At e_3, we cannot use $\Delta(\langle p_2 \rangle)$ to define $\Delta(\langle p_1, p_2 \rangle)$ since $\mathsf{disable}(\langle p_1 \rangle) > \mathcal{T}(\langle p_2 \rangle)$. Moreover, we cannot use $\Delta(\langle p_1 \rangle)$ to define $\Delta(\langle p_1, p_2 \rangle)$, either, because $\Delta(\langle p_2 \rangle)$ was defined before $\Delta(\langle p_1 \rangle)$ but was not used to create $\Delta(\langle p_1, p_2 \rangle)$ at e_1 due to the use of the enable set, indicating that the trace slice for $\langle p_1, p_2 \rangle$ does not belong to $\mathcal{G}$, and it should be ignored during monitoring. This intuition can be captured as the following condition: $\mathcal{T}(\langle p_2 \rangle) < \mathcal{T}(\langle p_1 \rangle)$ and $\langle p_2 \rangle \not\sqsubseteq \langle p_1 \rangle$. To reiterate, if $\Delta(\theta')$ is used to define $\Delta(\theta)$ and there exists some $\theta'' \sqsubset \theta$ s.t. $\theta'' \not\sqsubseteq \theta'$ and $\mathcal{T}(\theta'') < \mathcal{T}(\theta')$, then the trace slice for θ does not belong to the desired category set $\mathcal{G}$, because θ would have been in the enable set of θ' if it were in $\mathcal{G}$. Such a situation happens at the following conditions: 1) a creation event, $e\langle\theta''\rangle$, is encountered before $\Delta(\theta')$ is defined at event e'; 2) e is omitted when $\Delta(\theta')$ is defined (otherwise $\Delta(\theta'' \sqcup \theta')$ should have been defined and should be used to define θ instead of θ'). The second condition implies that $\mathsf{Dom}(\theta'') \notin \mathsf{enable}^X_{\mathcal{G}}(e')$. Therefore, when we combine θ'' and θ' in θ, the trace slice for θ cannot belong to $\mathcal{G}$, due to the definition of enable set.

Event	$e_2\langle p_2 \rangle$	$e_1\langle p_1 \rangle$	$e_3\langle p_1, p_2 \rangle$
Δ	$\langle p_2 \rangle : \sigma(i, e_2)$	$\langle p_2 \rangle : \sigma(i, e_2)$ $\langle p_1 \rangle : \sigma(i, e_1)$	$\langle p_2 \rangle : \sigma(i, e_2)$ $\langle p_1 \rangle : \sigma(i, e_1)$
$\mathcal{T}$	$\langle p_2 \rangle : 1$	$\langle p_2 \rangle : 1$ $\langle p_1 \rangle : 3$	$\langle p_2 \rangle : 1$ $\langle p_1 \rangle : 3$
disable	$\langle p_2 \rangle : 2$	$\langle p_2 \rangle : 2$ $\langle p_1 \rangle : 4$	$\langle p_2 \rangle : 2$ $\langle p_1 \rangle : 4$ $\langle p_1, p_2 \rangle : 5$

Figure 13. Another Monitoring Using Enable Sets and Timestamps.

Based on the above discussion, we develop a new parametric monitoring algorithm that optimizes algorithm $\mathbb{C}^+\langle X \rangle$ using the enable set and timestamps, as shown in Figure 14. This algorithm makes use of the mappings discussed above, namely, $\mathsf{enable}^X_{\mathcal{G}}$, Δ, $\mathcal{U}$, disable and $\mathcal{T}$, and maintains an integer variable to track the timestamp. Similar to algorithm $\mathbb{C}^+\langle X \rangle$, when event $e\langle\theta\rangle$ is received, algorithm $\mathbb{D}\langle X \rangle$ first checks whether $\Delta(\theta)$ is defined or not (line 1 in main). If not, monitor states may be generated for new encountered parameter instances, which is achieved by function createNewMonitorStates in algorithm $\mathbb{D}\langle X \rangle$. Unlike in algorithm $\mathbb{C}^+\langle X \rangle$, where all the parameter instances less informative than θ are searched to find all the compatible parameter instances using $\mathcal{U}$, createNewMonitorStates enumerates parameter sets in $\mathsf{enable}^X_{\mathcal{G}}(e)$ and looks for parameter instances

```
Algorithm 𝔻⟨X⟩(M = (S, ℰ, 𝒞, ι, σ, γ))
Input: mapping enable^X_𝒢 : [ℰ →° 𝒫_f(𝒫_f(X))]
Globals: mapping Δ : [[X →° V] →° S]
         mapping 𝒯 : [[X →° V] →° integer]
         mapping 𝒰 : [X →° V] → 𝒫_f([X →° V])
         mapping disable : [[X →° V] →° integer]
         integer timestamp
Initialization: 𝒰(θ) ← ∅ for any θ, timestamp ← 0

function main(e⟨θ⟩)
 1 if Δ(θ) undefined then
 2 ⋮ createNewMonitorState(e⟨θ⟩)
 3 ⋮ if Δ(θ) undefined and e is a creation event then
 4 ⋮ ⋮ defineNew(θ)
 5 ⋮ endif
 6 ⋮ disable(θ) ← timestamp
 7 ⋮ timestamp ← timestamp + 1
 8 endif
 9 foreach θ' ∈ {θ} ∪ 𝒰(θ) s.t. Δ(θ') defined do
10 ⋮ Δ(θ') ← σ(Δ(θ'), e)
11 endfor
function createNewMonitorStates(e⟨θ⟩)
 1 foreach X_e ∈ enable^X_𝒢(e)
   (in reversed topological order) do
 2 ⋮ if Dom(θ) ⊈ X_e then
 3 ⋮ ⋮ θ_m ← θ' s.t. θ' ⊏ θ and Dom(θ') = Dom(θ) ∩ X_e
 4 ⋮ ⋮ foreach θ'' ∈ 𝒰(θ_m) ∪ {θ_m} s.t. Dom(θ'') = X_e do
 5 ⋮ ⋮ ⋮ if Δ(θ'') defined and Δ(θ'' ⊔ θ) undefined then
 6 ⋮ ⋮ ⋮ ⋮ defineTo(θ'' ⊔ θ, θ'')
 7 ⋮ ⋮ ⋮ endif
 8 ⋮ ⋮ endfor
 9 ⋮ endif
10 endfor
function defineNew(θ)
 1 foreach θ'' ⊏ θ do
 2 ⋮ if Δ(θ'') defined then return endif
 3 endfor
 4 Δ(θ) ← ι
 5 𝒯(θ) ← timestamp
 6 timestamp ← timestamp + 1
 7 foreach θ'' ⊏ θ do
 8 ⋮ 𝒰(θ'') ← 𝒰(θ'') ∪ {θ}
 9 endfor
function defineTo(θ, θ')
 1 foreach θ'' ⊑ θ s.t. θ'' ⋤ θ' do
 2 ⋮ if disable(θ'') > 𝒯(θ') or 𝒯(θ'') < 𝒯(θ') then
 3 ⋮ ⋮ return
 4 ⋮ endif
 5 endfor
 6 Δ(θ) ← Δ(θ')
 7 𝒯(θ) ← 𝒯(θ')
 8 foreach θ'' ⊏ θ do
 9 ⋮ 𝒰(θ'') ← 𝒰(θ'') ∪ {θ}
10 endfor
```

Figure 14. Optimized Monitoring Algorithm $\mathbb{D}\langle X\rangle$.

whose domains are in $\mathsf{enable}^X_{\mathcal{G}}(e)$ and which are compatible with θ, also using $\mathcal{U}$. The inclusion check at line 2 in createNewMonitorStates is to omit unnecessary search since if $\mathsf{Dom}(\theta) \subseteq X_e$ then no new parameter instance will be created from θ. This way, createNewMonitorStates creates all the parameter instances that combine θ with compatible parameter instances that also satisfy the enable set of e using fewer lists in $\mathcal{U}$.

If e is a creation event then a monitor state for θ is initialized (lines 3 - 5 in main). Note that $\Delta(\theta)$ can be defined in function createNewMonitorStates if $\Delta(\theta')$ has been defined for some $\theta' \sqsubset \theta$. $\mathsf{disable}(\theta)$ is set to the current timestamp after all the creations and the timestamp is increased. The rest of function main in $\mathbb{D}\langle X\rangle$ is the same as in $\mathbb{C}^+\langle X\rangle$: all the relevant monitor states are updated according to e.

Function defineNew in $\mathbb{D}\langle X\rangle$ is similar to the one in $\mathbb{C}^+\langle X\rangle$. The only difference is that $\mathcal{T}(\theta)$ is set to the current timestamp, and the timestamp is incremented. Function defineTo in $\mathbb{D}\langle X\rangle$ checks disable and $\mathcal{T}$ as discussed above to decide whether $\Delta(\theta)$ can be defined using $\Delta(\theta')$. If $\Delta(\theta)$ is defined using $\Delta(\theta')$, $\mathcal{T}(\theta)$ is set to $\mathcal{T}(\theta')$.

In all of our tested cases $\mathbb{D}\langle X\rangle$ performs better than $\mathbb{C}^+\langle X\rangle$; in fact, in most cases that caused notable monitoring overhead, the efficiency of $\mathbb{D}\langle X\rangle$ is significantly better than $\mathbb{C}^+\langle X\rangle$. For example, in two extreme cases, $\mathbb{C}^+\langle X\rangle$ could not finish, while $\mathbb{D}\langle X\rangle$ had no problems. In terms of memory usage $\mathbb{D}\langle X\rangle$ also performs better, as expected, except for a few cases where $\mathbb{C}^+\langle X\rangle$ generates more garbage collections, reducing peak memory usage at the expense of performance.

6.2 Proofs of Correctness

The goal of this section is to show that algorithms $\mathbb{D}\langle X\rangle$ and $\mathbb{C}^+\langle X\rangle$ produce the same mapping Δ for the same given trace. $\mathbb{C}\langle X\rangle$ is already known to be correct for our definition of parametric trace monitoring due to the results in (9). As mentioned $\mathbb{C}^+\langle X\rangle$ is a straight-forward extension of $\mathbb{C}\langle X\rangle$. Thus by showing that $\mathbb{D}\langle X\rangle$ and $\mathbb{C}^+\langle X\rangle$ produce the same Δ (i.e., showing that they behave the same), we show that $\mathbb{D}\langle X\rangle$, itself, is correct for our definition of parametric monitoring.

We fix a trace $\tau = e_1e_2...e_n$, a monitor $M = (S, \mathcal{E}, \mathcal{C}, \iota, \sigma, \gamma)$ and a desired value set $\mathcal{G}$ in what follows. We use $\Delta_{\mathbb{C}}$ and $\Delta_{\mathbb{D}}$ to refer to the Δ in algorithms $\mathbb{C}^+\langle X\rangle$ and $\mathbb{D}\langle X\rangle$, respectively. For convenience, we also let $\mathsf{timestamp} : [\mathsf{integer} \overset{\circ}{\to} \mathsf{integer}]$ be the function defined as follows: $\mathsf{timestamp}(k)$ is the value of timestamp in $\mathbb{D}\langle X\rangle$ at the event e_k for $0 < k \leq n$; otherwise $\mathsf{timestamp}(k)$ is undefined. timestamp and $\mathcal{T}$ in $\mathbb{D}\langle X\rangle$ have the following properties:

PROPOSITION 3. *The follow holds for timestamp and $\mathcal{T}$ used in algorithm $\mathbb{D}\langle X\rangle$.*

1. *For $0 < k, k' \leq n$, $k \geq k'$ iff $\mathsf{timestamp}(k) \geq \mathsf{timestamp}(k')$.*
2. *$\Delta_{\mathbb{D}}(\theta)$ is defined iff $\mathcal{T}(\theta)$ is defined.*

Proof. 1. is obvious since timestamp is monotonic along the observed trace. 2. holds because $\Delta_{\mathbb{D}}(\theta)$ and $\mathcal{T}(\theta)$ are always defined together (lines 1 and 2 in defineNew and lines 6 and 7 in defineTo). □

We next define two functions that describe *when* and *how* a monitor state is created for a parameter instance.

DEFINITION 14. *Function set* : $[[X \overset{\circ}{\rightarrow} V] \overset{\circ}{\rightarrow}$ *integer*$]$ *is defined as follows:* $set(\theta) = k$ *if* $\Delta(\theta)$ *is initialized at* e_k. *Function* $MT : [[X \overset{\circ}{\rightarrow} V] \overset{\circ}{\rightarrow} [X \overset{\circ}{\rightarrow} V]^*]$ *is defined as follows:* $MT(\theta) = \theta_1...\theta_m$ *where* $\theta_m = \theta$, θ_1 *is initialized with* $\imath$, *and* $\Delta(\theta_i)$ *is initialized using* $\Delta(\theta_{i-1})$ *at some event* e *for any* $1 < i \leq m$.

Obviously, for both $\mathbb{C}^+\langle X\rangle$ and $\mathbb{D}\langle X\rangle$, $\mathsf{set}(\theta)$ is defined if and only if $\mathsf{MT}(\theta)$ is defined. Let $\mathsf{set}_{\mathbb{C}}$ and $\mathsf{set}_{\mathbb{D}}$ be the set in algorithm $\mathbb{C}^+\langle X\rangle$ and $\mathbb{D}\langle X\rangle$, respectively, and let $\mathsf{MT}_{\mathbb{C}}$ and $\mathsf{MT}_{\mathbb{D}}$ be the MT in algorithm $\mathbb{C}^+\langle X\rangle$ and $\mathbb{D}\langle X\rangle$, respectively.

PROPOSITION 4. *For algorithms* $\mathbb{C}^+\langle X\rangle$ *and* $\mathbb{D}\langle X\rangle$, *the following hold for set and MT:*

1. *For* θ_i *and* θ_j *in* $MT(\theta)$, $\theta_i \sqsubset \theta_j$ *if* $i < j$.
2. *If* $MT_{\mathbb{D}}(\theta) = \theta_1...\theta_m$ *then* $\mathcal{T}(\theta) = timestamp(set_{\mathbb{D}}(\theta_1))$.
3. *If* $set_{\mathbb{D}}(\theta)$ *is defined then* $set_{\mathbb{C}}(\theta)$ *is defined and* $set_{\mathbb{C}}(\theta) \leq set_{\mathbb{D}}(\theta)$.
4. *If* $set_{\mathbb{C}}(\theta) = set_{\mathbb{D}}(\theta)$ *and* $\Delta_{\mathbb{C}}(\theta) = \Delta_{\mathbb{D}}(\theta)$ *when they are initialized, then* $\Delta_{\mathbb{C}}(\theta) = \Delta_{\mathbb{D}}(\theta)$ *during the whole monitoring process.*
5. *If* $set_{\mathbb{C}}(\theta) = set_{\mathbb{D}}(\theta)$ *and* $MT_{\mathbb{C}}(\theta) = MT_{\mathbb{D}}(\theta)$ *then* $\Delta_{\mathbb{C}}(\theta) = \Delta_{\mathbb{D}}(\theta)$ *during the whole monitoring process.*

Proof.
1. It follows by Definition 14 and line 6 in createNewMonitorStates in $\mathbb{D}\langle X\rangle$.

2. Prove by induction on the length of $\mathsf{MT}_{\mathbb{D}}(\theta)$. If $\mathsf{MT}_{\mathbb{D}}(\theta) = \theta$, suppose that $\Delta_{\mathbb{D}}(\theta)$ is defined at event e_k, i.e., $\mathsf{set}_{\mathbb{D}}(\theta) = k$. Obviously, $\Delta_{\mathbb{D}}(\theta)$ is defined using defineNew in $\mathbb{D}\langle X\rangle$. Hence, $\mathcal{T}(\theta) = \mathsf{timestamp}(k)$ according to line 2 in defineNew. Now suppose that for $0 < j$ and any θ'' s.t. $\mathsf{MT}_{\mathbb{D}}(\theta'') = \theta_1...\theta_m$ and $m < j$, $\mathcal{T}(\theta'') = \mathsf{timestamp}(\mathsf{set}_{\mathbb{D}}(\theta_1))$. If $\mathsf{MT}_{\mathbb{D}}(\theta) = \theta_1...\theta_j$ then $\theta = \theta_j$ and $\Delta_{\mathbb{D}}(\theta)$ is defined using $\Delta_{\mathbb{D}}(\theta_{j-1})$ by Definition 14. $\mathcal{T}(\theta_j) = \mathcal{T}(\theta_{j-1})$ according to line 7 in defineTo in $\mathbb{D}\langle X\rangle$. By induction, $\mathcal{T}(\theta) = \mathcal{T}(\theta_{j-1}) = \mathsf{timestamp}(\mathsf{set}_{\mathbb{D}}(\theta_1))$.

3. Prove by induction on the length of $\mathsf{MT}_{\mathbb{D}}(\theta)$. We only need to show that if $\Delta_{\mathbb{D}}(\theta)$ is defined at event e_k and $\Delta_{\mathbb{C}}(\theta)$ is undefined before e_k then $\Delta_{\mathbb{C}}(\theta)$ is defined at e_k. If $\mathsf{MT}_{\mathbb{D}}(\theta) = \theta$, suppose $\mathsf{set}_{\mathbb{D}}(\theta) = k$ and $e_k\langle\theta'\rangle$. Since θ is not initialized with another parameter instance, it should be defined using defineNew function in $\mathbb{D}\langle X\rangle$, which only occurs via line 4 in main. Hence, $\theta' = \theta$ and e_k is a creation event. If $\Delta_{\mathbb{C}}(\theta)$ is undefined before e_k, it will be defined at e_k because line 10 in the main function in $\mathbb{C}^+\langle X\rangle$ will be executed if $\Delta_{\mathbb{C}}(\theta)$ is undefined before line 9.

Now suppose that for any parameter instance θ'' s.t. $\mathsf{set}_{\mathbb{D}}(\theta'')$ is defined and the length of $\mathsf{MT}_{\mathbb{D}}(\theta'')$ is less than j, $\mathsf{set}_{\mathbb{C}}(\theta'') \leq \mathsf{set}_{\mathbb{D}}(\theta'')$. If $\mathsf{set}_{\mathbb{D}}(\theta)$ is defined and $\mathsf{MT}_{\mathbb{D}}(\theta) = \theta_1...\theta_j$ where $\theta_j = \theta$, let $\mathsf{set}_{\mathbb{D}}(\theta) = k$ and $e_k\langle\theta'\rangle$. By Definition 14, $\Delta_{\mathbb{D}}(\theta)$ is defined using $\Delta_{\mathbb{D}}(\theta_{j-1})$. Hence, $\mathsf{set}_{\mathbb{D}}(\theta_{j-1}) < k$ and $\theta' \sqcup \theta_{j-1} = \theta$ according to line 6 in the createNewMonitorStates function in $\mathbb{D}\langle X\rangle$. By induction, $\mathsf{set}_{\mathbb{C}}(\theta_{j-1}) \leq \mathsf{set}_{\mathbb{D}}(\theta_{j-1}) < k$, that is, $\Delta_{\mathbb{C}}(\theta_{j-1})$ is defined before e_k. Therefore, if $\Delta_{\mathbb{C}}(\theta)$ is undefined before e_k, $\Delta_{\mathbb{C}}(\theta_j)$ will be defined in $\mathbb{C}^+\langle X\rangle$ at e_k because: if $\theta' = \theta$ then $\Delta_{\mathbb{C}}(\theta)$ will be defined at line 8 in main in $\mathbb{C}^+\langle X\rangle$ ($\theta_{j-1} \sqsubset \theta$ by 1.); otherwise, it will be defined at line 15 in main ($\theta' \sqcup \theta_{j-1} = \theta$).
4. In both $\mathbb{C}^+\langle X\rangle$ and $\mathbb{D}\langle X\rangle$, after $\Delta(\theta)$ is defined at e_k, it will be updated using any event $e_j\langle\theta'\rangle$ with $\theta' \sqsubseteq \theta$ and $k < j$. If $\mathsf{set}_{\mathbb{C}}(\theta) = \mathsf{set}_{\mathbb{D}}(\theta)$ and $\mathsf{MT}_{\mathbb{C}}(\theta) = \mathsf{MT}_{\mathbb{D}}(\theta)$ then $\mathsf{MT}_{\mathbb{C}}(\theta)$ and $\mathsf{MT}_{\mathbb{D}}(\theta)$ will be updated using the same events afterward and therefore equivalent during the whole monitoring.
5. It can be easily proved by induction on the length of $\mathsf{MT}_{\mathbb{D}}(\theta)$ and 4.

□

The following lemma shows that $\mathbb{C}^+\langle X\rangle$ and $\mathbb{D}\langle X\rangle$ are equivalent for monitors that are created from the initial state.

LEMMA 1. *The following hold for MT:*

1. *If* $MT_{\mathbb{C}}(\theta) = \theta$ *then* $MT_{\mathbb{D}}(\theta) = \theta$ *and* $set_{\mathbb{C}}(\theta) = set_{\mathbb{D}}(\theta)$.
2. *If* $MT_{\mathbb{D}}(\theta) = \theta$ *then* $MT_{\mathbb{C}}(\theta) = \theta$ *and* $set_{\mathbb{C}}(\theta) = set_{\mathbb{D}}(\theta)$.

Proof.
1. If $\mathsf{MT}_{\mathbb{C}}(\theta) = \theta$, suppose that $\mathsf{set}_{\mathbb{C}}(\theta) = k$. Obviously, $\Delta_{\mathbb{C}}(\theta)$ is defined by the defineNew function in $\mathbb{C}^+\langle X\rangle$, which only occurs when e_k is a creation event and comes with the parameter instance θ. Also, for all $\theta' \sqsubset \theta$, $\Delta_{\mathbb{C}}(\theta')$ is undefined before e_k; otherwise, $\Delta_{\mathbb{C}}(\theta)$ should be defined using $\Delta_{\mathbb{C}}(\theta')$ at line 8 in main in $\mathbb{C}^+\langle X\rangle$. By Proposition 4 3., $\Delta_{\mathbb{D}}(\theta)$ and $\Delta_{\mathbb{D}}(\theta')$, for all $\theta' \sqsubset \theta$, are undefined before e_k. So $\Delta_{\mathbb{D}}(\theta)$ cannot be defined in the createNewMonitorStates function in $\mathbb{D}\langle X\rangle$ using some $\theta' \sqsubset \theta$ when e_k is encountered. Hence, the condition at line 3 in main in $\mathbb{D}\langle X\rangle$ is satisfied and line 4 will be executed to initialize $\Delta_{\mathbb{D}}(\theta)$ using defineNew in $\mathbb{D}\langle X\rangle$. Therefore, $\mathsf{MT}_{\mathbb{D}}(\theta) = \theta$ and $\mathsf{set}_{\mathbb{D}}(\theta) = k = \mathsf{set}_{\mathbb{C}}(\theta)$.

2. By Proposition 4.3., if $\mathsf{MT}_{\mathbb{D}}(\theta) = \theta$ and $\mathsf{set}_{\mathbb{D}}(\theta) = k$ then $\mathsf{MT}_{\mathbb{C}}(\theta)$ is defined before or at e_k. Assume that $\mathsf{MT}_{\mathbb{C}}(\theta) = \theta_1..\theta_m$ and $m > 1$. Then we have 1) $\theta_1 \sqsubset \theta$ by Proposition 4 1.; 2) $\mathsf{MT}_{\mathbb{D}}(\theta_1) = \mathsf{MT}_{\mathbb{C}}(\theta_1) = \theta_1$ and $\mathsf{set}_{\mathbb{C}}(\theta_1) = \mathsf{set}_{\mathbb{D}}(\theta_1)$ 1.; 3) $\mathsf{set}_{\mathbb{C}}(\theta_1) < \mathsf{set}_{\mathbb{C}}(\theta) \leq \mathsf{set}_{\mathbb{D}}(\theta)$ by Proposition 4.3. Let $e_k\langle\theta'\rangle$. Since $\mathsf{MT}_{\mathbb{D}}(\theta) = \theta$, $\Delta_{\mathbb{D}}(\theta)$ is defined using defineNew via line 4 in main in $\mathbb{D}\langle X\rangle$ when e_k is encountered. Hence, $\theta = \theta'$. However, since $\Delta_{\mathbb{D}}(\theta_1)$ is defined before e_k, the condition at line 2 in defineNew is satisfied and $\Delta_{\mathbb{D}}(\theta)$ cannot be defined at e_k. Contradiction reached. Therefore, $\mathsf{MT}_{\mathbb{C}}(\theta) = \theta$. By 1., $\mathsf{set}_{\mathbb{C}}(\theta) = \mathsf{set}_{\mathbb{D}}(\theta)$. □

PROPOSITION 5. *For algorithms* $\mathbb{C}^+\langle X\rangle$ *and* $\mathbb{D}\langle X\rangle$, *the following hold:*

1. *If* $MT_{\mathbb{C}}(\theta) = MT_{\mathbb{D}}(\theta)$ *then for any* $\theta' \in MT_{\mathbb{C}}(\theta)$, $set_{\mathbb{C}}(\theta') = set_{\mathbb{D}}(\theta')$.
2. *If* $MT_{\mathbb{C}}(\theta) = MT_{\mathbb{D}}(\theta)$ *then* $\Delta_{\mathbb{C}}(\theta) = \Delta_{\mathbb{D}}(\theta)$ *during the whole monitoring.*

Proof.
1. Suppose $\mathsf{MT}_{\mathbb{C}}(\theta) = \theta_1, .., \theta_m$. Prove by induction on $\mathsf{MT}_{\mathbb{C}}(\theta)$. For θ_1, since $\mathsf{MT}_{\mathbb{C}}(\theta_1) = \theta_1$, $\mathsf{set}_{\mathbb{C}}(\theta_1) = \mathsf{set}_{\mathbb{D}}(\theta_1)$ by Lemma 1.1. Now suppose that for some $1 < jleqm$, $\mathsf{set}_{\mathbb{C}}(\theta_i) = \mathsf{set}_{\mathbb{D}}(\theta_i)$ for any $0 < i < j$. Assume that $\mathsf{set}_{\mathbb{C}}(\theta_j) \neq \mathsf{set}_{\mathbb{D}}(\theta_j)$. We have $\mathsf{set}_{\mathbb{C}}(\theta_j) < \mathsf{set}_{\mathbb{D}}(\theta_j)$ by Proposition 4.3. Let $\mathsf{set}_{\mathbb{C}}(\theta_j) = k$ and $e_k\langle\theta''\rangle$. Since $\theta'' \sqcup \theta_{j-1} = \theta_j$, we have $\theta'' \not\sqsubset \theta_{j-1}$. Also, $\mathsf{disable}(\theta'') > \mathsf{timestamp}(k) > \mathcal{T}(\theta_{j-1})$ after e_k. Let $\mathsf{set}_{\mathbb{D}}(\theta) = g$. We have that $\Delta_{\mathbb{D}}(\theta_j)$ cannot be defined at e_g using $\Delta_{\mathbb{D}}(\theta_{j-1})$ because $g > k$ and θ'' will satisfy the condition at line 2 in defineTo in $\mathbb{D}\langle X\rangle$. Contradiction found. Therefore, $\mathsf{set}_{\mathbb{C}}(\theta_j) = \mathsf{set}_{\mathbb{D}}(\theta_j)$.

2. Follow by 1. and Proposition 4.5. □

Let $\Delta^{\tau}_{\mathbb{C}}$ be the Δ after $\mathbb{C}^+\langle X\rangle$ processes τ and $\Delta^{\tau}_{\mathbb{D}}$ be the Δ after $\mathbb{D}\langle X\rangle$ processes τ.

PROPOSITION 6. *The following holds:*

1. *If* $\gamma(\Delta^{\tau}_{\mathbb{C}}(\theta)) \in \mathcal{G}$ *and for any* $\theta_i \in MT_{\mathbb{C}}(\theta)$, $i > 1$, *let* $set_{\mathbb{C}}(\theta_i) = k$, *we have* $Dom(\theta_{i-1}) \in enable^X_{\mathcal{G}}(e_k)$.
2. *If* $\gamma(\Delta^{\tau}_{\mathbb{C}}(\theta)) \in \mathcal{G}$ *then* $MT_{\mathbb{C}}(\theta) = MT_{\mathbb{D}}(\theta)$.

Proof.
1. Suppose that the sliced trace for θ is $\tau_\theta = e'_1\langle\theta'_1\rangle...e'_h\langle\theta'_h\rangle$. Then $\sigma(\tau_\theta) = \Delta^{\tau}_{C}(\theta)$, according to Theorem 3 in (9). Since $\gamma(\Delta^{\tau}_{\mathbb{C}}(\theta)) \in \mathcal{G}$, $P(\tau_\theta) \in \mathcal{G}$. Also, since $\Delta_{\mathbb{C}}(\theta_i)$ is defined at e_k, $e_k \in \tau_\theta$ and it is the first occurrence of e_k in τ_θ. Suppose that e'_n is the first occurrence of e_k in τ_θ. Then $\mathsf{enable}_{\tau}(e_k) = \{e'_1, ..., e'_{n-1}\}$ by Definition 11. For any $0 < j < n$, let $e'_j\langle\theta''\rangle$, then $\theta'' \sqsubseteq \theta_{i-1}$; otherwise, e'_j should not be contained in the slice for θ_{i-1} and thus not in the slice for θ_i (since $\Delta_{\mathbb{C}}(\theta_i)$ is initialized using $\Delta_{\mathbb{C}}(\theta_{i-1})$.) Hence, $\cup^X_{\{e'_1,...,e'_{n-1}\}} = \mathsf{Dom}(\theta_{i-1})$, that is, $\mathsf{Dom}(\theta_{i-1}) \in \mathsf{enable}^X_{\mathcal{G}}(e_k)$ by Definition 13.

2. Suppose that $\mathsf{MT}_{\mathbb{C}}(\theta) = \theta_1, ..., \theta_m$. Prove by induction on $\mathsf{MT}_{C}(\theta)$. For θ_1, $\mathsf{MT}_{\mathbb{C}}(\theta_1) = \theta_1$. Hence, $\mathsf{MT}_{\mathbb{D}}(\theta_1) = \theta_1$ by Lemma 1. Now suppose that for some $1 < j \leq m$, we have $\mathsf{MT}_{\mathbb{D}}(\theta_{j-1}) = \mathsf{MT}_{C}(\theta_{j-1}) = \theta_1, ..., \theta_{j-1}$. Let $\mathsf{set}_{\mathbb{C}}(\theta_j) = k$ and $e_k\langle\theta'\rangle$. By Proposition 4 3., $\Delta_{\mathbb{D}}(\theta_j)$ is undefined before e_k. Also, $\theta' \sqcup \theta_{j-1} = \theta_j$ due to line 15 in main in $\mathbb{C}^+\langle X\rangle$.

By 1., $\mathsf{Dom}(\theta_{j-1}) \in \mathsf{enable}^X_{\mathcal{G}}(e_k)$. Hence, $\Delta_{\mathbb{D}}(\theta_j)$ will be defined at e_k because of the loop from line 4 - 8 in createNewMonitorStates in $\mathbb{D}\langle X\rangle$. We only need to show that $\Delta_{\mathbb{D}}(\theta_j)$ is defined using $\Delta_{\mathbb{D}}(\theta_{j-1})$. Assume that $\Delta_{\mathbb{D}}(\theta_j)$ is defined using $\Delta_{\mathbb{D}}(\theta'')$ and $\theta'' \neq \theta_{j-1}$. Then we have $\theta'' \sqcup \theta' = \theta_j$. $\theta'' \not\sqsubset \theta_{j-1}$ because the loop from line 1 to line 10 in createNewMonitorStates in $\mathbb{D}\langle X\rangle$ is carried out in a reverse topological order. Also, $\theta_{j-1} \not\sqsubset \theta''$ because the loops from line 2 to line 6 and from line 12 to line 18 in main in $\mathbb{C}^+\langle X\rangle$ are carried out in a reverse topological order. Such situation, i.e., θ_j does not have a maximum sub-instance, is impossible according to the proof for algorithm $\mathbb{A}\langle X\rangle$ in (9). Contradiction found. Therefore, $\Delta_{\mathbb{D}}(\theta_j)$ is defined using $\Delta_{\mathbb{D}}(\theta_{j-1})$ at e_k. We then have $\mathsf{MT}_{\mathbb{D}}(\theta_j) = \mathsf{MT}_{\mathbb{D}}(\theta_{j-1})\theta_j = \mathsf{MT}_{\mathbb{C}}(\theta_{j-1})\theta_j = \mathsf{MT}_{\mathbb{C}}(\theta_j)$. By induction, $\mathsf{MT}_{\mathbb{C}}(\theta_m) = \mathsf{MT}_{\mathbb{D}}(\theta_m)$.

□

PROPOSITION 7. *If* $\Delta^{\tau}_{\mathbb{D}}(\theta)$ *is defined then* $MT_{\mathbb{C}}(\theta) = MT_{\mathbb{D}}(\theta)$.

Proof. Suppose that $\mathsf{MT}_{\mathbb{D}}(\theta) = \theta_1, ..., \theta_m$. Prove by induction on $\mathsf{MT}_{\mathbb{D}}(\theta)$. For θ_1, $\mathsf{MT}_{\mathbb{D}}(\theta_1) = \theta_1$. Hence, $\mathsf{MT}_{\mathbb{C}}(\theta_1) = \theta_1$ by Lemma 1.2. Now suppose that for some $1 < j \leq m$, we have $\mathsf{MT}_{\mathbb{D}}(\theta_{j-1}) = \mathsf{MT}_{C}(\theta_{j-1}) = \theta_1, ..., \theta_{j-1}$. Let $\mathsf{set}_{\mathbb{D}}(\theta_j) = k$ and $e_k\langle\theta'\rangle$.

Suppose that $\mathsf{MT}_{\mathbb{C}}(\theta_j) = \theta^j_1...\theta^j_h$ where $\theta^j_h = \theta_j$. We first show that $\theta_1 = \theta^j_1$ by contradiction. Assume $\theta_1 \neq \theta^j_1$. Let $\mathsf{set}_{\mathbb{C}}(\theta^j_1) = p^j$ and $\mathsf{set}_{\mathbb{D}}(\theta_1) = p$. Since $\mathsf{MT}_{\mathbb{C}}(\theta^j_1) = \theta^j_1$ and $\mathsf{MT}_{\mathbb{D}}(\theta_1) = \theta_1$, we have that $e_{p^j}\langle\theta^j_1\rangle$, $e_p\langle\theta_1\rangle$ and they are both creation events. We also have $\mathcal{T}_{\mathbb{D}}(\theta_1) = \mathsf{timestamp}(p)$. By Proposition 4.2, $\Delta_{\mathbb{D}}(\theta^j_1)$ is not defined before p^j. Hence, $\Delta_{\mathbb{D}}(\theta^j_1)$ is defined at p^j and $\mathcal{T}_{\mathbb{D}}(\theta^j_1) = \mathsf{timestamp}(p^j)$. Also, $\mathsf{disable}(\theta^j_1) > \mathcal{T}_{\mathbb{D}}(\theta^j_1)$ since line 6 in main of algorithm $\mathbb{D}\langle X\rangle$ is executed after $\mathcal{T}_{\mathbb{D}}(\theta^j_1)$ is defined at line 4. Since $\theta_1 \neq \theta^j_1$, $p^j \neq p$; in other words, either $p^j < p$ or $p^j > p$. Therefore, either $\mathcal{T}_{\mathbb{D}}(\theta^j_1) < \mathcal{T}_{\mathbb{D}}(\theta_1)$ or $\mathcal{T}_{\mathbb{D}}(\theta_1) < \mathcal{T}_{\mathbb{D}}(\theta^j_1) < \mathsf{disable}(\theta^j_1)$ by Proposition 3.1. Let θ_n be the first parameter instance in $\mathsf{MT}_{\mathbb{D}}(\theta_j)$ s.t. $\theta^j_1 \sqsubset \theta_n$ and $\theta^j_1 \not\sqsubset \theta_{n-1}$, $n > 1$, and let $\mathsf{set}_{\mathbb{D}}(\theta_n) = p_n$. Then $\Delta_{\mathbb{D}}(\theta_n)$ is defined in the defineTo function in $\mathbb{D}\langle X\rangle$ at e_{p_n} using $\Delta_{\mathbb{D}}(\theta_{n-1})$. However, it is impossible since θ^j_1 satisfies the condition at line 2 in defineTo and prevents defining $\Delta_{\mathbb{D}}(\theta_n)$ at e_{p_n}. Contradiction found and $\theta_1 = \theta^j_1$.

Assume that $\mathsf{MT}_{\mathbb{C}}(\theta_j) \neq \mathsf{MT}_{\mathbb{D}}(\theta_j)$. We can find $l > 1$ s.t. $\theta^j_l \neq \theta_l$ and $\theta^j_i = \theta_i$ for any $0 < i < l$. Let $\mathsf{set}_{\mathbb{C}}(\theta^j_l) = k$ and $\mathsf{set}_{\mathbb{C}}(\theta_l) = g$. Suppose $e_{n_l}\langle\theta''\rangle$. We have $\theta^j_{l-1} \sqcup \theta'' = \theta^j_l$; so $\theta'' \not\sqsubset \theta^j_{l-1}$. Also, $\mathsf{disable}(\theta'') > \mathcal{T}(\theta^j_l) = \mathcal{T}(\theta^j_1) = \mathcal{T}(\theta_1)$ after e_k. $k < g$ is impossible; otherwise, $\Delta_{\mathbb{D}}(\theta_l)$ cannot be defined at e_g using $\Delta_{\mathbb{D}}(\theta_{l-1})$ because θ'' will satisfy the condition at line 2 in defineTo in $\mathbb{D}\langle X\rangle$. Hence, $k > g \geq \mathsf{set}_{\mathbb{C}}(\theta_l)$ by Proposition 4.3. In other words, $\Delta_{\mathbb{C}}(\theta_l)$ is defined before e_k. Therefore, $\theta_l \notin \mathsf{MT}_{\mathbb{C}}(\theta_j)$ but $\theta_l \subseteq \theta_j$. Then we can find $\theta^j_p \in \mathsf{MT}_{\mathbb{C}}(\theta_j)$ s.t. $\theta_l \sqsubset \theta^j_p$ and $\theta_l \not\sqsubset \theta_i$ for any $0 < i < p$. However, suppose $\mathsf{set}_{\mathbb{C}}(\theta^j_p) = n$, then at event e_n, we have $\theta_l \sqsubset \theta^j_p$ and $\theta_l \not\sqsubset \theta^j_{p-1}$. According to the proof for algorithm $\mathbb{A}\langle X\rangle$ in (9), we should have $\theta^j_{p-1} \sqsubset \theta_l$, which means that $\Delta_{\mathbb{C}}(\theta^j_p)$ should be defined using $\Delta_{\mathbb{C}}(\theta_l)$. Contradiction found. Therefore, $\mathsf{MT}_{\mathbb{C}}(\theta_j) = \mathsf{MT}_{\mathbb{D}}(\theta_j)$.

□

THEOREM 1. *The following holds:*

1. *if* $\gamma(\Delta^{\tau}_{\mathbb{C}}(\theta)) \in \mathcal{G}$ *then* $\gamma(\Delta^{\tau}_{\mathbb{D}}(\theta)) = \gamma(\Delta^{\tau}_{\mathbb{C}}(\theta))$;
2. *if* $\gamma(\Delta^{\tau}_{\mathbb{D}}(\theta)) \in \mathcal{G}$ *then* $\gamma(\Delta^{\tau}_{\mathbb{C}}(\theta)) = \gamma(\Delta^{\tau}_{\mathbb{D}}(\theta))$;

3. $\gamma(\Delta^{\tau}_{\mathbb{C}}(\theta)) \in \mathcal{G}$ *iff* $\gamma(\Delta^{\tau}_{\mathbb{D}}(\theta)) = \gamma(\Delta^{\tau}_{\mathbb{C}}(\theta))$ *iff* $\gamma(\Delta^{\tau}_{\mathbb{D}}(\theta)) \in \mathcal{G}$.

Proof.
1. By Proposition 6 and Proposition 5.2, $\Delta^{\tau}_{\mathbb{D}}(\theta) = \Delta^{\tau}_{\mathbb{C}}(\theta)$. Hence, $\gamma(\Delta^{\tau}_{\mathbb{D}}(\theta)) = \gamma(\Delta^{\tau}_{\mathbb{C}}(\theta))$.

2. Follow by Proposition 7 and Proposition 5.2.

3. Follow by 1 and 2. □

Theorem 1 states that a trace slice for θ is reported by $\mathbb{C}^{+}\langle X\rangle$ to be in $\mathcal{G}$ if and only if it is also reported by $\mathbb{D}\langle X\rangle$ to be in $\mathcal{G}$. In other words,$\mathbb{C}^{+}\langle X\rangle$ and $\mathbb{D}\langle X\rangle$ are equivalent for those parameter instances whose trace slices are in $\mathcal{G}$. Thus $\mathbb{D}\langle X\rangle$ is complete and sound.

7. Implementation and Evaluation

We implemented code generation for Algorithms $\mathbb{C}^{+}\langle X\rangle$ and $\mathbb{D}\langle X\rangle$ in JavaMOP. The indexing technique proposed in (8) is used to implement all the mappings in the algorithms. Some optimizations were applied to $\mathbb{D}\langle X\rangle$ to achieve more efficient monitoring code. First, a method is generated for each event and in the method, $\mathsf{enable}^{X}_{\mathcal{G}}(e)$ is statically determined. Thus the main loop in createNewMonitorStates is unrolled in the monitoring code. In every iteration of the unrolled loop, X_e is statically determined. Hence, the condition at line 2 and θ_m at line 3 in createNewMonitorStates can be statically computed for each iteration and the resulting values are constants in the generated code. The invocation of function defineTo at line 6 in createNewMonitorStates is statically expanded using the function body of defineTo in every unrolled iteration of the main loop. This way, the context information of call sites can be used to optimize every copy of the defineTo function. For example, the domains of θ and θ' are fixed in each iteration of the unrolled loop in createNewMonitorStates, so we can also unroll the loop from line 1 to line 5 in defineTo and compute the comparison between θ' and θ'' at code generation time. Also, the inner loop (lines 4 - 8) in createNewMonitorStates checks every parameter instance in $\mathcal{U}(\theta)$ but $\mathcal{U}(\theta)$ may contain many other instances whose domains are not X_e. To reduce runtime overhead, the code generation makes a mapping for each e and $X_e \in \mathsf{enable}^{X}_{\mathcal{G}}(e)$. Specifically, given an event definition $\mathcal{D}_{\mathcal{E}}$, for any event e and every $X_e \in \mathsf{enable}^{X}_{\mathcal{G}}(e)$, a mapping $\mathcal{U}^{e}_{X_e}$ is generated to map the parameter instance θ_m with $\mathsf{Dom}(\theta_m) = \mathcal{D}_{\mathcal{E}}(\theta) \cap X_e$ to a list of parameter instances more informative than θ_m whose domain is X_e. In every iteration of the unrolled loop in createNewMonitorStates, the corresponding $\mathcal{U}^{e}_{X_e}$ is used for the inner loop. This way, fewer parameter instances are enumerated at runtime.

We evaluated $\mathbb{C}^{+}\langle X\rangle$, $\mathbb{D}\langle X\rangle$, and Tracematches on the DaCapo benchmark suite (4). We omitted other runtime systems because they have been evaluated and compared with either Tracematches or the original JavaMOP algorithm[5] in other papers (1; 2; 8). Note that Soot (18), the underlying bytecode engine for Tracematches, cannot handle the DaCapo benchmark properly, resulting in fewer instrumentation points in the pmd program. Accordingly, we modify our specification to have the same scope of instrumentation for a fair comparison. The raw results can be found at (17).

Experimental Settings. Our experiments were performed on a machine with 2GB RAM and a Pentium 4 2.66GHz processor. The machine's operating system is UBuntu Linux 7.10, and we used version 2006-10 of the DaCapo benchmark suite (4). The default input for DaCapo was used, and we use the -converge option to ensure the validity of our test by running each test multiple times, until the execution time converges. After convergence, the runtime is stabilized within 3%, thus numbers in Figure 15 should be interpreted as "±3%". Additional code introduced by the AspectJ weaving process changes the program structure in DaCapo; sometimes this causes the benchmark to run a little bit faster due to better instruction cache layout.

Properties. We used the following properties in our experiments. They were borrowed from (5; 6; 16).

- UnsafeMapIterator: Do not update a Map when using the Iterator interface to iterate its values or its keys;
- SafeSyncCollection: If a Collection is synchronized, then its iterator also should be accessed in a synchronized manner;
- SafeSyncMap: If a Collection is synchronized, then its iterators on values and keys also should be accessed in a synchronized manner;
- SafeIterator: Do not update a Collection when using the Iterator interface to iterate its elements;
- SafeFile: All file opens should be closed strictly in the function where it is opened;
- SafeFileWriter: No write to a FileWriter after closing.

UnsafeMapIterator, SafeSyncCollection, SafeSyncMap and SafeFile could not be monitored using the original JavaMOP algorithm, as they contain creation events that do not instantiate all the parameters. SafeFile and SafeFileWriter cannot be expressed in Tracematches because they are context-free properties. We use them to demonstrate the effectiveness of the enable set optimization on CFG properties. SafeIterator was chosen because it has generated some of the largest runtime overheads in previous experiments (8; 16).

Results and Discussions. Figure 15 summarizes the results of our experiments. It shows the percent overheads of $\mathbb{C}^{+}\langle X\rangle$, $\mathbb{D}\langle X\rangle$ (both implemented in JavaMOP), and Tracematches. All the properties were heavily monitored in the experiments. Millions of parameter instances were observed for some

[5] The original JavaMOP is conservatively extended by the new $\mathbb{D}\langle X\rangle$, in that for properties supported by the original JavaMOP algorithm, $\mathbb{D}\langle X\rangle$ generates the same monitoring code and instrumentation.

	UnsafeMapIterator			SafeSyncCollection			SafeSyncMap			SafeIterator			SafeFile		SafeFileWriter	
	TM	$\mathbb{C}^+\langle X\rangle$	$\mathbb{D}\langle X\rangle$	TM	$\mathbb{C}^+\langle X\rangle$	$\mathbb{D}\langle X\rangle$	TM	$\mathbb{C}^+\langle X\rangle$	$\mathbb{D}\langle X\rangle$	TM	$\mathbb{C}^+\langle X\rangle$	$\mathbb{D}\langle X\rangle$	$\mathbb{C}^+\langle X\rangle$	$\mathbb{D}\langle X\rangle$	$\mathbb{C}^+\langle X\rangle$	$\mathbb{D}\langle X\rangle$
antlr	-2	5	2	-2	2	1	-3	2	1	0	0	0	11	9	2	5
bloat	>10000	OOM	935	1448	735	712	2267	858	660	11258	769	749	3	1	5	0
chart	-1	4	0	0	1	1	1	3	0	11	5	3	-1	0	-1	0
eclipse	8	2	1	0	0	0	0	1	1	2	0	1	2	2	1	2
fop	11	-2	-3	-4	-3	0	16	-5	-3	5	4	1	-3	-3	-3	-5
hsqldb	29	0	0	24	0	0	22	-1	0	17	-1	0	2	0	0	-1
jython	57	11	7	6	-4	-4	8	-4	-5	16	-2	0	-4	-4	-3	-5
luindex	7	12	5	0	1	1	3	1	4	9	3	5	22	21	-1	-1
lusearch	9	1	-1	9	1	1	8	2	-1	34	4	2	0	-1	-1	0
pmd	>10000	OOM	196	33	18	15	50	21	12	196	19	14	-2	0	-2	-2
xalan	10	4	4	7	-1	1	6	0	0	10	9	8	0	-1	2	1

Figure 15. Average *Percent* Runtime Overhead for Tracematches(TM), $\mathbb{C}^+\langle X\rangle$, and $\mathbb{D}\langle X\rangle$ (convergence within 3%, OOM = Out of Memory). $\mathbb{D}\langle X\rangle$, at its worst, has less than an order of magnitude of overhead.

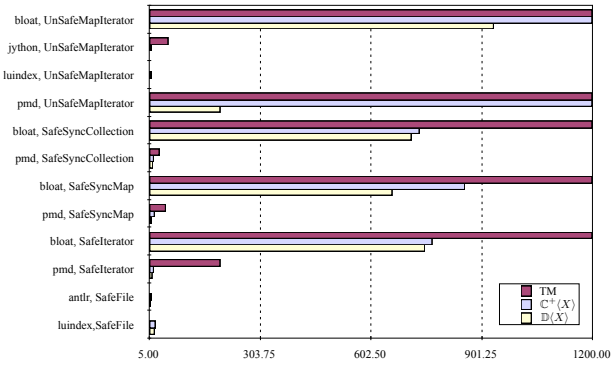

Figure 16. Runtime Overhead Statistics.

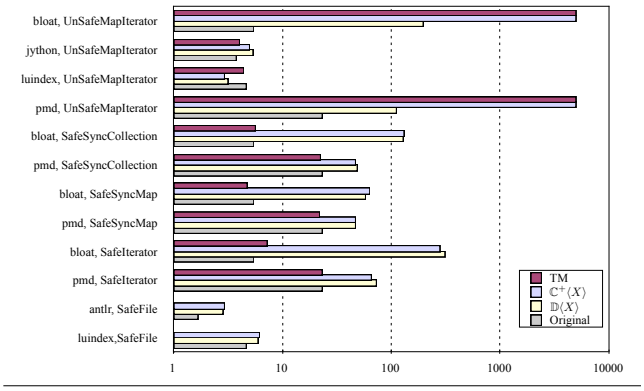

Figure 17. Peak Memory Usage Statistics.

properties under monitoring, e.g., SafeIterator, putting a critical test on the generated monitoring code. All three systems generated low runtime overhead in most experiments, showing their efficiency. For $\mathbb{D}\langle X\rangle$, only 7 out of 66 cases caused more than 10% runtime overhead. The numbers for $\mathbb{C}^+\langle X\rangle$ and Tracematches are 9 out of 66 and 15 out of 44, respectively. Figure 16 shows the comparison among three systems using the cases where at least two of them generated more than 10% overhead. In all cases, $\mathbb{D}\langle X\rangle$ outperformed the other two and $\mathbb{C}^+\langle X\rangle$ is better than Tracematches. This shows that JavaMOP provides an efficient solution to monitor parametric specifications despite its genericity in terms of specification formalisms.

The results also illustrate the effectiveness of the proposed optimization based on the enable set: on average, the overhead of $\mathbb{D}\langle X\rangle$ is about 20% less than $\mathbb{C}^+\langle X\rangle$. Moreover, when the property to monitor becomes more complicated, the improvement achieved by the optimization is more significant. In the two extreme cases, namely, bloat-UnsafeMapIterator and pmd-UnsafeMapIterator, where both the non-optimized JavaMOP and Tracematches crashed, the optimized JavaMOP managed to finish the executions with overheads that are reasonable for many applications, such as testing and debugging.

Figure 17 shows the maximum memory usages of our experiments in log_{10} scale, in Megabytes. Only the cases where significant overhead was incurred are shown.

Two interesting observations can be made from Figure 17. First, the enable set optimization does not always reduce

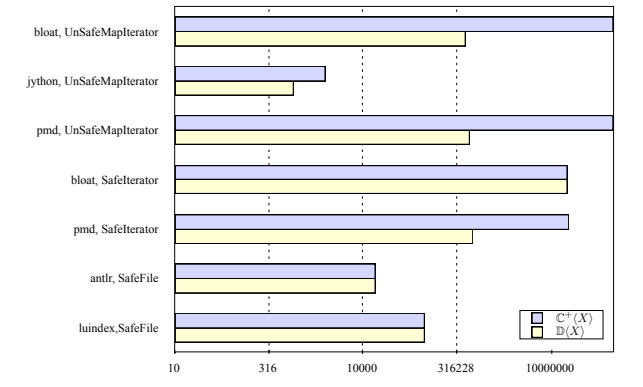

Figure 18. Number of Parameter Instance Monitors.

peak memory usage. In 7 out of 12 cases, the $\mathbb{D}\langle X\rangle$ has lower peak memory usage than $\mathbb{C}^+\langle X\rangle$. As mentioned, bloat-SafeMapIterator and pmd-SafeMapIterator, where $\mathbb{C}^+\langle X\rangle$ did not finish execution, exiting with an out of memory error, $\mathbb{D}\langle X\rangle$ managed to complete. In the other 5 cases, $\mathbb{C}^+\langle X\rangle$ has slightly lower peak memory usage than $\mathbb{D}\langle X\rangle$. This is because $\mathbb{C}^+\langle X\rangle$ caused more garbage collections than $\mathbb{D}\langle X\rangle$. For example, in pmd-SafeSyncCollection, the $\mathbb{C}^+\langle X\rangle$ had 1791 young generation garbage collections while $\mathbb{D}\langle X\rangle$ had 1465 collections. However, fewer garbage collection cycles contribute to the performance increase of the enable set optimization. The reduced memory usage of luindex-UnsafeMapIterator in comparison to the base program can

also be attributed to an increase in the number of garbage collections, 3857 for the base program, and 3957 and 4260 for the optimized and non-optimized JavaMOP runs, respectively. Another observation is that memory usage is *not directly related* to the resulting monitoring overhead: little memory overhead was observed for bloat-SafeSyncCollection and bloat-SafeSyncMap, which caused huge runtime overhead, while for pmd-SafeIterator, we observed high memory usage but insignificant runtime overhead.

Figure 18 shows the number of created monitor instances, another important measurement for our approach, also in log_{10} scale. Only 7 cases are shown because others generated many fewer monitor instances. For the experiments with significant runtime overhead. In bloat-SafeMapIterator and pmd-SafeMapIterator, the number of instance monitors in $\mathbb{C}^+\langle X\rangle$ is not precise, due to the out of memory crash. In all cases, the optimized JavaMOP generated an equal or lesser number of monitors than the non-optimized one, showing that the optimization is effective in reducing the number of monitors, particularly in the cases where many instance monitors are created. Also, the results indicate that the number of created monitor instances is not the only factor influencing runtime overhead: no monitor instances were created for bloat-SafeSyncCollection and bloat-SafeSyncMap, but they generated significant monitoring overhead. A further inspection revealed that in both cases, a tremendous number of related events were observed: 137880368 and 165269166 events for bloat-SafeSyncCollection and bloat-SafeSyncMap, respectively. This resulted in intensive monitoring work even when no monitor instances were created. Static *program* analysis may provide a better solution in such cases, which we plan to explore in our future work.

8. Conclusion

Efficient monitoring of parametric properties is a very challenging problem, due to the potentially huge number of parameter instances. Until now, solutions to this problem have either used a hardwired logical formalism, or limited their handling of parameters. Our approach, based on a general semantics of parametric traces with a property-based optimization, overcomes these limitations, without sacrificing any efficiency, as our evaluation shows.

References

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*, pages 345–364. ACM, 2005.

[2] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, pages 589–608. ACM, 2007.

[3] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*, pages 44–57, 2004.

[4] 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 *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'06)*, pages 169–190. ACM, 2006.

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

[6] 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, 2007.

[7] F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Runtime Verification (RV'03)*, volume 89 of *ENTCS*, pages 108–127, 2003.

[8] F. Chen and G. Roşu. MOP: An efficient and generic runtime verification framework. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, pages 569–588. ACM, 2007.

[9] F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 246–261, 2009. extended version: UIUCDCS-R-2008-2954.

[10] Temporal Rover. `http://www.time-rover.com`.

[11] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*, pages 385–402. ACM, 2005.

[12] K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In *Runtime Verification (RV'01)*, volume 55 of *ENTCS*, 2001.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *European Conference on Object Oriented Programming (ECOOP'01)*, volume 2072 of *LNCS*, pages 327–353, 2001.

[14] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a runtime assurance tool for Java. In *Runtime Verification (RV'01)*, volume 55 of *ENTCS*, 2001.

[15] M. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Object Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, pages 365–383. ACM, 2005.

[16] P. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. In *Automated Software Engineering (ASE '08)*, pages 148–157. IEEE, 2008.

[17] Javamop generic parametric monitoring results. http://fsl.cs.uiuc.edu/JavaMOP_OOPSLA_Results.

[18] Soot website. http://www.sable.mcgill.ca/soot/.