Efficient Formalism-Independent Monitoring of Parametric Properties

Feng Chen and Dongyun Jin and Patrick O'Neil Meredith and Grigore Roşu
University of Illinois at Urbana-Champaign
Urbana, Illinois,USA 61801
fengchen, djin3, pmeredit, grosu@illinois.edu

Abstract

Efficient monitoring of parametric properties, in spite of increasingly growing interest thanks 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. A few solutions have been proposed, but most of them compromise their expressiveness for performance or vice versa. In this paper, we propose a generic, in terms of specification formalisms, yet efficient, solution to monitoring parametric specifications. Our approach is based on a general semantics for slicing parametric traces and makes use of knowledge about the property to monitor. The needed knowledge is not specific to the underlying formalism and can be easily computed when generating monitoring code from the property. An extensive evaluation shows that the monitoring code generated by our algorithm is still faster than 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 [11, 13, 9, 6, 8, 1, 14, 10, 7], 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, is receiving more and more interest due to its effectiveness in handling system behaviors that involve multiple instances of different components. Let us discuss the following example about using classes Map, Collection and Iterator in Java Util library.

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 being used to enumerate elements in the map, content of the map should not be changed or unexpected behaviors may occur. This property 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: $\forall m, c, i.\mathsf{create\_coll}(m, c) \rightarrow \mathsf{not} \diamond (\mathsf{create\_iter}(c, i) \wedge \diamond(\mathsf{update\_map}(m) \wedge \diamond \mathsf{use\_iter}(i)))$, 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 specification states that if Collection c is obtained from a Map m then it implies in the future, the following should not happen: an iterator i is created from c and then m is changed and then i is accessed.

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 $\mathsf{createIter}(s, i)$ is received, we need to find all $\mathsf{getSet}(m, s)$ events with the same binding for s.

Several approaches were introduced to support specifying and monitoring of parametric specifications, including Eagle [8], Tracematches [1], PQL [14], PTQL [10] and MOP [7]. But they are all limited in terms of underlying specification formalisms or supported properties. Eagle, Tracematches, PQL and PTQL adopt logic-specific solutions, namely, each of them hardwires a specification formalism, e.g., regular patterns or context-free patterns, extends the formalism with parameters and then develops an algorithm to generate monitoring code for the extended but still particular formalism. Although this approach provides a solution to monitoring parametric specifications, we ar-

```
SafeMapIterator(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*(..))) && target(m) ||
        (call(* Map.put*(..)) || call(* Map.putAll*(..)) || call(* Map.clear())) {}
  cfg : S -> create_coll create_iter Nexts update_map Updates use_iter,
        Nexts -> Nexts use_iter | epsilon,
        Updates -> Updates update_map | update_map
    @validation{ System.out.println("cfg: Accessed Invalid Iterator!"); }
  ere : create_coll update_map* create_iter use_iter* update_map update_map* use_iter
    @validation{ System.out.println("ere: Accessed Invalid Iterator!"); }
  ptltl: use_iter /\ <*> (create_iter /\ <*> create_coll) -> (not update_map) S create_iter
    @violation{ System.out.println("ptltl: Accessed Invalid Iterator!"); }
  ftltl: create_coll -> not <> (create_iter /\ <> (update_map /\ <> use_iter))
    @violation{ System.out.println("ftltl: Accessed Invalid Iterator!"); }
}
```

Figure 1: CFG, ERE, PTLTL, and FTLTL SafeMapIterator

gue 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, our experiments show that our formalism-independent solution generates more efficient monitoring code than the very efficient Tracematches.

MOP, on the other hand, adopts a divide-and-conquer solution. It does not fix the formalism to use in the specification, instead, MOP provides an efficient and generic framework for monitoring of parametric specifications, which allows one to use existing non-parametric formalisms in parametric specifications. Figure 1 shows a JavaMOP2[1] specification that specifies the above Map-Set-Iterator property using four[2] different formalisms, namely, context-free grammar (CFG), extended regular expressions (ERE), past-time linear temporal logic (PTLTL) and future-time linear temporal logic (FTLTL). In Figure 1, we first name the specified property and give the parameters used in the specification. Then we define the involved events using an AspectJ-like syntax, for example, create_coll is defined as the returning of functions values and keyset of Map. Every event may instantiate some parameters at runtime. For example, create_coll will instantiate parameters m and c using the target and the return value of the method call.

Non-parametric patterns/formulae are provided in separated sections following the event definition using different formalisms. Every section starts with the used formalism, e.g., ere for ERE and ftltl for FTLTL, and then consists of a pattern/formula of defined events together with *handlers* which will be executed when the specified property is validated or violated. In Figure 1, the CFG and ERE patterns describe behavioral patterns violating the property; so they are monitored for validations and associated with @validation handlers that will be executed when the monitored execution matches the pattern. Conversely, the PTLTL and FTLTL formulae state the desired property directly and are therefore monitored for violations. MOP provides the user the flexibility to use the most convenient formalism to specify the property. For example, as shown in Figure 1, FTLTL describes the desired property more concisely, but ERE may be easier to understand for a programmer.

However, the original technique used in MOP restricted the type of properties allowed for monitoring, to ensure the correctness and efficiency of monitoring. More specifically, only properties whose *creation events*, i.e., events triggering the monitoring, such as the getSet event in the above example, define the complete parameter binding for the specification were allowed. Hence, MOP could not handle a large set of properties, including the above Map-Set-Iterator pattern.

Contributions: We present a general technique to build optimized parametric monitors from non-parametric monitors, following the spirit of MOP but *without the limitation*. The technique is based on theoretical results in [16]. As our experiments show, a straight implementation of the conceptual algorithm in [16] causes a prohibitive runtime. Hence, we introduce an optimization to apply knowledge about the monitored property to avoid handling redundant parameter bindings. The needed knowledge, event *enable sets*, depends only on the property and not on the formalism used to specify it. We show that the enable set information is easily achieved as a "side effect" when generating a non-parametric monitor from the property by extending existing monitoring generation algorithms for ERE, PTLTL, FTLTL and CFG. Our technique has been implemented in the latest version of JavaMOP. An extensive evaluation on JavaMOP

[1] JavaMOP2 is a novel implementation of the MOP paradigm[7] for Java. The main difference between JavaMOP2 and its predecessor here called JavaMOP1 is that JavaMOP2 supports the algorithm discussed in this paper in its full generality. In this paper we, however, focus on the underlying generic technique and algorithms, not on JavaMOP2.

[2] Only one of them would suffice; as it is, the MOP spec in Figure 1 reports four messages whenever the property is violated. We show all four of them for two reasons: (1) to emphasize the formalism-independence of our approach; and (2) we refer to them later in the paper.

shows that the proposed technique significantly improves the efficiency of monitoring. Monitoring code generated by our approach performs even better than other approaches optimized for fixed formalisms. For simplicity, in the rest of this paper, we use JavaMOP2 to refer to the implementation of the technique presented in this paper and JavaMOP1 for the previous limited implementation.

2 Semantics of 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 [16]. We include only the core definitions here to make this paper self-contained.

2.1 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, don't know traces, etc.). Parametric properties are parametric trace classifiers and provide, for each parameter instance, the category to which the trace slice corresponding to that parameter instance belongs. Trace slicing is defined as a reduct operation that forgets all the events that are unrelated to the given parameter instance.

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. For example, for the regular pattern in Figure 1, create_coll create_iter update_map use_iter is a validating 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. Note that ftltl and ptltl formulae, for example, do not have don't know traces.

Definition 3 ***(Parametric events and traces).*** *Let X be a set of **parameters** and let V_X be a set of corresponding **parameter values**. If $\mathcal{E}$ is a set of base events like in Definition 1, then let $\mathcal{E}\langle X\rangle$ denote the set of corresponding **parametric events** $e\langle\theta\rangle$, where e is a base event in $\mathcal{E}$ and θ is* **parameter instance***, i.e., a partial map in $[X \stackrel{\circ}{\to} V_X]$. 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 ={m, c, i} is a set of parameters (of types {Map, Collection, Iterator}, respectively) and $V_X = \{m_1, c_1, i_1, i_2\}$, then create_coll$\langle$m $\mapsto m_1$, c $\mapsto c_1\rangle$, create_iter$\langle$c$\mapsto c_1$, i$\mapsto i_1\rangle$, and use_iter$\langle$i$\mapsto i_1\rangle$, are parametric events and create_coll$\langle$m$\mapsto m_1$, c$\mapsto c_1\rangle$ create_iter$\langle$c$\mapsto c_1$, i$\mapsto i_1\rangle$ use_iter$\langle$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 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 parameter that will be instantiated by e at runtime. Parametric event $e\langle\theta\rangle$ is an **instance** of event definition $\mathcal{D}_\mathcal{E}$ if $\mathsf{Dom}(\theta) = \mathcal{D}_\mathcal{E}(e)$. Parametric trace τ **follows** $\mathcal{D}_\mathcal{E}$ if $e\langle\theta\rangle$ is an instance of $\mathcal{D}_\mathcal{E}$ for any $e\langle\theta\rangle \in \tau$.*

For example, (create_coll $\mapsto$ {m, s}, create_iter $\mapsto$ {s, i}, use_iter $\mapsto$ {i}, update_map $\mapsto$ {m}) is a parametric event definition that corresponds to the example in Figure 1. It states that two parameters, namely, m and s, will be instantiated at runtime when a parametric event create_coll$\langle\theta\rangle$ is received, and so on. create_coll$\langle$m$\mapsto m_1$, s$\mapsto s_1\rangle$ 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$\langle m_1, s_1\rangle$ instead of create_coll$\langle$m$\mapsto m_1$, s$\mapsto s_1\rangle$, and $\langle m_1, s_1\rangle$ instead of $\langle$m$\mapsto m_1$, s$\mapsto s_1\rangle$.

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

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

With the notation above, $\langle m_1, s_1\rangle$ and $\langle s_1, i_1\rangle$ are compatible and $\langle m_1, s_1\rangle \sqcup \langle s_1, i_1\rangle = \langle m_1, s_1, i_1\rangle$.

Definition 6 ***(Trace slicing)*** *Given parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$ and θ in $[X \stackrel{\circ}{\to} V_X]$, 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*
- $(\tau\, e\langle\theta'\rangle)\restriction_\theta = \begin{cases} (\tau\restriction_\theta)\, e & \text{when } \theta' \sqsubseteq \theta \\ \tau\restriction_\theta & \text{when } \theta' \not\sqsubseteq \theta \end{cases}$,

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

Consider the parametric trace create_coll$\langle m_1, s_1\rangle$ create_iter$\langle s_1, i_1\rangle$ use_iter$\langle i_1\rangle$ update_map$\langle m_1\rangle$, the trace slice for $\langle m_1\rangle$ is update_map. The trace slice for $\langle m_1, s_1\rangle$ is create_coll update_map, and the trace slice for $\langle m_1, s_1, i_1\rangle$ is create_coll create_iter use_iter update_map.

Definition 7 *Let* X *be a set of parameters with their corresponding values* V_X*, 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* $[[X \overset{\circ}{\to} V_X] \to \mathcal{C}]$*)*

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

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

$\Lambda X.P$ 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.

2.2 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 in our context. Even though only a finite number of states is reached during any given (finite) execution trace, there is, in general, no bound on how many. For example, monitors for context-free grammars like the ones in [15] have potentially unbounded stacks as part of their state. Also, as shown shortly, parametric monitors have domains of functions as state spaces, which are infinite as well. 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* $\gamma(\sigma(1, w)) = P(w)$ *for each* $w \in \mathcal{E}^*$.

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* V_X *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*

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

with $\Lambda X.\sigma : [[X \overset{\circ}{\to} V_X] \to S] \times \mathcal{E}\langle X\rangle \to [[X \overset{\circ}{\to} V_X] \to S]$ *and* $\Lambda X.\gamma : [[X \overset{\circ}{\to} V_X] \to S] \to [[X \overset{\circ}{\to} V_X] \to \mathcal{C}]$ *defined as*

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

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

for any $\delta \in [[X \overset{\circ}{\to} V_X] \to S]$ *and any* $\theta, \theta' \in [X \overset{\circ}{\to} V_X]$.

Parametric monitor $\Lambda X.M$ is a monitor for the parametric property $\Lambda X.P$, with P the property monitored by each non-parametric monitor M[16].

3 Online Parametric Monitoring

In this section we discuss algorithms for efficient online monitoring of parametric properties, given a non-parametric monitor M. We start with a base algorithm that extends algorithm $\mathbb{C}\langle X\rangle$ in [16] to support *creation events*. Then we show that it can be significantly improved provided that certain knowledge about the specified property is available. Most formal definitions and proofs are omitted due to limited space and can be found in the companion report [5].

3.1 Utilizing Creation Events

The first challenge to online monitoring of a parametric property is that the state space of potential parameter instances is infinite. Like in [16], we encode functions $[[X \overset{\circ}{\to} V_X] \overset{\circ}{\to} Y]$, which map parameter instances to some values, as tables with entries indexed by parameter instances in $[X \overset{\circ}{\to} V_X]$ and with contents values in Y. It can be proved that such tables will have a finite number of entries provided that each event instantiates finite parameters.

Figure 2 then gives the $\mathbb{C}^+\langle X\rangle$ algorithm for online monitoring of parametric property $\Lambda X.P$, given that M is the 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.

Algorithm $\mathbb{C}^+\langle X\rangle(M = (S, \mathcal{E}, \mathcal{C}, 1, \sigma, \gamma))$

Globals: mapping $\Delta : [[X \xrightarrow{\circ} V_X] \xrightarrow{\circ} S]$
mapping $\mathcal{U} : [X \xrightarrow{\circ} V_X] \to \mathcal{P}_f([X \xrightarrow{\circ} V_X])$

Initialization: $\mathcal{U}(\theta) \leftarrow \emptyset$ for any $\theta \in [X \xrightarrow{\circ} V_X]$

function main($e\langle\theta\rangle$)
1 if $\Delta(\theta)$ undefined then
2 ⋮ foreach $\theta_m \sqsubset \theta$ (in reversed topological order)do
3 ⋮ ⋮ if $\Delta(\theta_m)$ defined then
4 ⋮ ⋮ ⋮ goto 7
5 ⋮ ⋮ endif
6 ⋮ endfor
7 ⋮ if $\Delta(\theta_m)$ defined then
8 ⋮ ⋮ defineTo(θ, θ_m)
9 ⋮ elseif e is a creation event then
10 ⋮ ⋮ defineNew(θ)
11 ⋮ endif
12 ⋮ foreach $\theta_m \sqsubset \theta$ (in reversed topological order)do
13 ⋮ ⋮ foreach $\theta_{comp} \in \mathcal{U}(\theta_m)$ compatible with θdo
14 ⋮ ⋮ ⋮ if $\Delta(\theta_{comp} \sqcup \theta)$ undefined then
15 ⋮ ⋮ ⋮ ⋮ defineTo($\theta_{comp} \sqcup \theta, \theta_{comp}$)
16 ⋮ ⋮ ⋮ endif
17 ⋮ ⋮ endfor
18 ⋮ endfor
19 endif
20 foreach $\theta' \in \{\theta\} \cup \mathcal{U}(\theta)$do
21 ⋮ $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e)$
22 endfor

function defineNew(θ)
1 $\Delta(\theta) \leftarrow 1$
2 foreach $\theta'' \sqsubset \theta$ do
3 ⋮ $\mathcal{U}(\theta'') \leftarrow \mathcal{U}(\theta'') \cup \{\theta\}$
4 endfor

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

Figure 2: Monitoring algorithm $\mathbb{C}^+\langle X\rangle$

It is a slightly different variant of algorithm $\mathbb{C}\langle X\rangle$ in [16]. The difference comes from an observation of monitoring in practice: 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 Figure1, we can choose to start monitoring on a pair of m and s objects,(m_1, s_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 *creation events*. Algorithm $\mathbb{C}^+\langle X\rangle$ extends $\mathbb{C}\langle X\rangle$ in [16] to support creation events. It is easy to show that $\mathbb{C}\langle X\rangle$ can be regarded as a special case of $\mathbb{C}^+\langle X\rangle$, that is, all the events are creation events.

Two mappings are used in this algorithm: 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 are properly more informative than θ. In what follows, "the monitor state for θ" is used to refer 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 $\Delta(\theta)$.

When parametric event $e\langle\theta\rangle$ is received, the algorithm first checks whether θ has been encountered yet by checking if its corresponding monitor state, i.e., $\Delta(\theta)$, is defined(line 1 in main). If θ is encountered for the first time, new parameter instances may need to be created. In such 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 to 6 in main). If such θ_m is found, its monitor state is used to initialize the monitor state for θ (lines 7 and 8 in main); otherwise, a new monitor state is created for θ *only if* e is a creation event (lines 9 and 10 in main). Also, new parameter instances can be created by combining θ with existing parameter instances that are compatible with θ, i.e., they don't 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 to 17 in main).

If θ has been seen before or after all the new monitor states have been created, 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 to 22 in main). The updates also make use of the lists stored in $\mathcal{U}$. 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 then update the lists in table $\mathcal{U}$ to maintain its integrity.

Event	update_map$\langle m_1 \rangle$	create_coll$\langle m_1, s_1 \rangle$	create_coll$\langle m_2, s_2 \rangle$	create_iter$\langle s_1, i_1 \rangle$
Δ	$\emptyset$	$\langle m_1, s_1 \rangle$: $\sigma(i, \text{create\_coll})$	$\langle m_1, s_1 \rangle$: $\sigma(i, \text{create\_coll})$ $\langle m_2, s_2 \rangle$: $\sigma(i, \text{create\_coll})$	$\langle m_1, s_1 \rangle$: $\sigma(i, \text{create\_coll})$ $\langle m_2, s_2 \rangle$: $\sigma(i, \text{create\_coll})$ $\langle m_1, s_1, i_1 \rangle$: $\sigma(\sigma(i, \text{create\_coll}), \text{create\_iter})$
$\mathcal{U}$	$\emptyset$	$\bot : \langle m_1, s_1 \rangle$ $\langle m_1 \rangle$: $\langle m_1, s_1 \rangle$ $\langle s_1 \rangle$: $\langle m_1, s_1 \rangle$	$\bot : \langle m_1, s_1 \rangle, \langle m_2, s_2 \rangle$ $\langle m_1 \rangle$: $\langle m_1, s_1 \rangle$ $\langle s_1 \rangle$: $\langle m_1, s_1 \rangle$ $\langle m_2 \rangle$: $\langle m_2, s_2 \rangle$ $\langle s_2 \rangle$: $\langle m_2, s_2 \rangle$	$\bot : \langle m_1, s_1 \rangle, \langle m_2, s_2 \rangle, \langle m_1, s_1, i_1 \rangle$ $\langle m_1 \rangle$: $\langle m_1, s_1 \rangle, \langle m_1, s_1, i_1 \rangle$ $\langle s_1 \rangle$: $\langle m_1, s_1 \rangle, \langle m_1, s_1, i_1 \rangle$ $\langle m_2 \rangle$: $\langle m_2, s_2 \rangle$ $\langle s_2 \rangle$: $\langle m_2, s_2 \rangle$ $\langle i_1 \rangle$: $\langle m_2, s_2 \rangle, \langle m_2, s_2, i_1 \rangle$ $\langle m_1, s_1 \rangle$: $\langle m_1, s_1, i_1 \rangle$ $\langle m_1, i_1 \rangle$: $\langle m_1, s_1, i_1 \rangle$ $\langle s_1, i_1 \rangle$: $\langle m_1, s_1, i_1 \rangle$

Table 1: An example run of $\mathbb{C}^+\langle X \rangle$.

It can be proved that $\mathbb{C}^+\langle X \rangle$ correctly creates and updates monitor states according to the received event. The proof can be easily derived from the proof for algorithm $\mathbb{C}\langle X \rangle$ in [16] and is omitted here. We next use an example run, illustrated in Table 1, to show how $\mathbb{C}^+\langle X \rangle$ works. In Table 1, we show the content 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$\langle m_1, s_1 \rangle$ create_coll$\langle m_2, s_2 \rangle$ create_iter$\langle s_1, i_1 \rangle$. 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$\langle m_1, s_1 \rangle$, is a creation event. So a new monitor state is defined in Δ for $\langle m_1, s_1 \rangle$, which is also added to the lists in $\mathcal{U}$ for $\bot$, $\langle m_1 \rangle$ and $\langle s_1 \rangle$. Note that $\bot$ is the parameter instance that contains no parameter binding. So it is less informative than any other parameter instances. The third event create_coll$\langle m_2, s_2 \rangle$ 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$\langle s_1, i_1 \rangle$ is not a creation event. So no monitor state is created for $\langle s_1, i_1 \rangle$. It is compatible with the existing parameter instance $\langle m_1, s_1 \rangle$ introduced by the second event but not compatible with $\langle m_2, s_2 \rangle$ due to the conflict binding of s. The compatible instance $\langle m_1, s_1 \rangle$ can be found from the list for $\langle s_1 \rangle$ in $\mathcal{U}$. Therefore, a new monitor state is defined for the combined parameter instance $\langle m_1, s_1, i_1 \rangle$ using the state for $\langle m_1, s_1 \rangle$ in Δ. $\mathcal{U}$ is also updated to add the combined parameter instance into lists of parameter instances that are less informative.

3.2 Overcoming the Limitations of $\mathbb{C}^+\langle X \rangle$

$\mathbb{C}^+\langle X \rangle$ 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 a monitor generation algorithm is also provided. However, this generality leads to extra monitoring overhead in some cases. Let us continue the run in Table 1 to process one more event, use_iter$\langle i_1 \rangle$. The result is showed in Table 2. use_iter$\langle i_1 \rangle$ is not a creation event and no monitor state is created for $\langle i_1 \rangle$. But since $\langle i_1 \rangle$ is compatible with $\langle m_2, s_2 \rangle$, a new monitor state is defined for $\langle m_2, s_2, i_1 \rangle$. And the monitor state for $\langle m_1, s_1, i_1 \rangle$ is updated according to use_iter because $\langle i_1 \rangle$ is less informative than $\langle m_1, s_1, i_1 \rangle$. $\mathcal{U}$ is also updated to include parameter instances less informative than $\langle m_2, s_2, i_1 \rangle$.

Event	use_iter$\langle i_1 \rangle$
Δ	$\langle m_1, s_1 \rangle$: $\sigma(i, \text{create\_coll})$ $\langle m_2, s_2 \rangle$: $\sigma(i, \text{create\_coll})$ $\langle m_1, s_1, i_1 \rangle$: $\sigma(\sigma(\sigma(i, \text{create\_coll}), \text{create\_iter}), \text{use\_iter})$ $\langle m_2, s_2, i_1 \rangle$: $\sigma(\sigma(i, \text{create\_coll}), \text{use\_iter})$
$\mathcal{U}$	$\bot : \langle m_1, s_1 \rangle, \langle m_2, s_2 \rangle, \langle m_2, s_2, i_1 \rangle, \langle m_1, s_1, i_1 \rangle$ $\langle m_1 \rangle$: $\langle m_1, s_1 \rangle, \langle m_1, s_1, i_1 \rangle$ $\langle s_1 \rangle$: $\langle m_1, s_1 \rangle, \langle m_1, s_1, i_1 \rangle$ $\langle m_2 \rangle$: $\langle m_2, s_2 \rangle, \langle m_2, s_2, i_1 \rangle$ $\langle s_2 \rangle$: $\langle m_2, s_2 \rangle, \langle m_2, s_2, i_1 \rangle$ $\langle i_1 \rangle$: $\langle m_2, s_2, i_1 \rangle, \langle m_1, s_1, i_1 \rangle$ $\langle m_2, \text{s} \mapsto s_2 \rangle$: $\langle m_2, s_2, i_1 \rangle$ $\langle m_2, \text{Iter} \mapsto i_1 \rangle$: $\langle m_2, s_2, i_1 \rangle$ $\langle s_2, \text{Iter} \mapsto i_1 \rangle$: $\langle m_2, s_2, i_1 \rangle$ $\langle m_1, \text{s} \mapsto s_1 \rangle$: $\langle m_1, s_1, i_1 \rangle$ $\langle m_1, \text{Iter} \mapsto i_1 \rangle$: $\langle m_1, s_1, i_1 \rangle$ $\langle s_1, \text{Iter} \mapsto i_1 \rangle$: $\langle m_1, s_1, i_1 \rangle$

Table 2: Following run of Table 1.

It is necessary to create the monitor state for $\langle m_2, s_2, i_1 \rangle$ if no specific knowledge about the property to monitor is provided. However, this particular monitor creation can be avoided when we monitor a specific property and are interested in only a certain subset of value categories ($\mathcal{C}$ in Definition 2). For example, suppose that the property to monitor is the regular expression in Figure 1, which depicts a defective interaction among related objects of m, s and i. To find an error in the program using monitoring is thus to match the execution with the pattern. It is obvious

that to match the pattern, for a parameter instance of parameter set {m, s, i}, create_coll and create_iter should be observed before use_iter is encountered for the first time. Otherwise, the corresponding trace slice will never match the pattern, i.e., the monitor state for the parameter instance will never reach the validation state. Taking this observation into account, we next show that the creation of the monitor state for $\langle m_2, s_2, i_1 \rangle$ in Table 2 is not needed. When event use_iter$\langle i_1 \rangle$ is encountered, if the monitor state for a parameter instance $\langle m_2, s_2 \rangle$ exists without the monitor state for $\langle m_2, s_2, i_1 \rangle$, like in Table 2, it can be inferred that in the trace slice for $\langle m_2, s_2, i_1 \rangle$, only event create_coll and update_map occur before use_iter (otherwise, the monitor state for $\langle m_2, s_2, i_1 \rangle$ would have been created). Therefore, no match of the specified pattern will be reached for $\langle m_2, s_2, i_1 \rangle$, making it unnecessary to create the monitor state for it. This way, the time and space cost of monitoring can be reduced because the size of Δ and $\mathcal{U}$ is decreased, and fewer parameter instances need to be examined afterward when more events are received.

Based on this observation, one may reduce the monitoring overhead by applying some knowledge about the property to monitor. We next formalize the needed information about the property, and argue that it is not specific to the underlying specification formalism, and that it can be computed easily. Using the information to optimize monitoring is discussed in Section 3.3.

Definition 11 ***(Trace enable set).*** *Given trace $\tau \in \mathcal{E}^*$ and $e, e' \in \tau$, we denote the fact that e' occurs before the first occurrence of e in τ as $e' \rightsquigarrow_\tau e$. Then we define the* ***trace enable set*** *of $e \in \mathcal{E}$ as a function $\mathsf{enable}_\tau : \mathcal{E} \rightarrow \mathcal{P}_f(\mathcal{E})$, as follows: $\mathsf{enable}_\tau(e) = \{e' \mid e' \rightsquigarrow_\tau e\}$.*

Note that if $e \notin \tau$ then $\mathsf{enable}_\tau(e) = \emptyset$. The trace enable set characterizes a trace and therefore can be used to quickly check the property category to which the trace may belong.

Definition 12 ***(Property enable set).*** *Given $P : \mathcal{E}^* \rightarrow \mathcal{C}$ and a set of categories $\mathcal{G} \subseteq \mathcal{C}$ as the goal, the* ***property enable set*** *of event $e \in \mathcal{E}$ is defined as a function $\mathsf{enable}_\mathcal{G}^{event} : \mathcal{E} \rightarrow \mathcal{P}_f(\mathcal{P}_f(\mathcal{E}))$ as follows: $\mathsf{enable}_\mathcal{G}^{event}(e) = \{\mathsf{enable}_\tau(e) | P(\tau) \in \mathcal{G}\}$.*

For example, given the regular pattern in Figure 1 we have $\mathsf{enable}_\mathcal{G}^{event}$(create_coll) = $\{\emptyset\}$, $\mathsf{enable}_\mathcal{G}^{event}$(create_iter) = $\{\{$create_coll$\}\}$, $\mathsf{enable}_\mathcal{G}^{event}$(use_iter) = $\{\{$create_coll, create_iter$\}$, $\{$create_coll, create_iter, update_map$\}\}$, and $\mathsf{enable}_\mathcal{G}^{event}$(update_map) = $\{\{$create_coll, create_iter$\}$, $\{$create_coll, create_iter, use_iter$\}\}$. The property enable set tells whether an incomplete trace slice has the possibility of reaching the desired categories or not by looking at the events that have already occurred. However, as shown in the above example, the monitoring process keeps track of encountered parameter instances and discards events after monitors consume them. It is more efficient than storing every event because the number of events can be enormous. Therefore, we need to adapt the notion of the property enable set to parameter sets.

For a set of parametric event definitions $\{e_1(X_1), ..., e_n(X_n)\}$, let $\cup^X_{\{e_1,...,e_n\}} = X_1 \cup ... \cup X_n$. Then we define the enable set using sets of parameters.

Definition 13 ***Property parameter enable set.*** *Given a property $P : \mathcal{E}^* \rightarrow \mathcal{C}$, a set of categories $\mathcal{G} \subseteq \mathcal{C}$ as the goal and a set of parameters X, the* ***property parameter enable set*** *of event $e \in \mathcal{E}$ is defined as a function $\mathsf{enable}_\mathcal{G} : \mathcal{E} \rightarrow \mathcal{P}_f(\mathcal{P}_f(X))$ as follows: $\mathsf{enable}_\mathcal{G}(e) = \{\cup^X_{\mathsf{enable}_\tau(e)} | P(\tau) \in \mathcal{G}\}$.*

From now, we use "enable set" to refer to "property parameter enable set" for simplicity. For example, given the regular pattern in Figure 1 and $\mathcal{T} = \{$validation$\}$, we have $\mathsf{enable}_\mathcal{G}$(create_coll) = $\{\emptyset\}$, $\mathsf{enable}_\mathcal{G}$(create_iter) = $\{\{$m, s$\}\}$, $\mathsf{enable}_\mathcal{G}$(use_iter) = $\{\{$m, s, i$\}\}$, and $\mathsf{enable}_\mathcal{G}$(update_map) = $\{\{$m, s, i$\}\}$. The following result shows that one can skip certain parameter instances using the enable set:

Proposition 1 *When algorithm $\mathbb{C}^+\langle X \rangle$ receives event $e\langle\theta\rangle$, if we use θ' to define $\theta \sqcup \theta'$ and $\mathsf{Dom}(\theta') \notin \mathsf{enable}_\mathcal{G}(e)$, then $\Delta(\theta \sqcup \theta') \notin \mathcal{G}$ during the whole monitoring process.*

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. For example, we develop an algorithm to compute the enable set for finite state automata based monitors, including monitors for ERE, PTLTL and FTLTL, using a similar technique proposed in [3] to find dependence among events. We also implement 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 [3].

Let $\mathcal{G} = \{$valid$\}$. For $\mathsf{enable}_\mathcal{G}^{event}$ and a given CFP $G = (NT, \Sigma, P, S)$ we begin with all productions $S \rightarrow \gamma$ and the set $\epsilon_0 = \emptyset \in \mathcal{P}_f(\mathcal{E})$. For each production, we investigate each $s \in \gamma$ from left to right. If $s \in \Sigma$ we add ϵ_i to $\mathsf{enable}_\mathcal{G}^{event}(s)$, thus if s is the first symbol in γ we add ϵ_0. We then add s to ϵ_i forming ϵ_{i+1}. If $s \in NT$ we recursively invoke the algorithm, but rather than use ϵ_0, we use ϵ_i, and each production investigated will be of the form $s \rightarrow \gamma$. We keep track of which $s \in NT$ have been processed, to ensure termination. $\mathcal{G} = \{$invalid$\}$ is handled in a similar manner,

[3] We assume a certain familiarity with context free patterns; definitions can be found in [15], together with explanations on CFG monitoring.

save that when $s \in \Sigma$, ϵ_i is added to all $\mathsf{enable}_{\mathcal{G}}^{event}(s')$ such that $s' \in \Sigma - s$.

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. While the computation of the enable set can be expensive, it represents static information about the given property and only need be generated once.

Discussions Other possibilities for optimization are exhibited in the example in Table 2. We discuss two of them here. The first solution is to make use of the semantics of the program. In this example, we know that an i object is created from a s object and does not relate to other s objects. Hence, we can avoid creating combination of $\langle m_2, s_2 \rangle$ and $\langle i_1 \rangle$ because i_1 is created from s_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 can and should be applied to further reduce the monitoring overhead, whenever it is available.

Another solution is 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 $\langle m_1, s_1 \rangle$ need to be combined to build a new parameter instance because s_1 and i_1 are connected in the second event, $\mathsf{create\_coll}\langle m_1, s_1 \rangle$, in Table 2, but $\langle i_1 \rangle$ and $\langle m_2, s_2 \rangle$ 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 of the limitation of online monitoring, i.e., no information about future events available, such a heuristic can break, for example, an event connecting the two parameter instances comes afterward. We believe that the enable set provides a sound solution and performs as well as, if not better than, such heuristics in most cases.

3.3 Enable Set Based Monitoring

Given a set of desired value categories $\mathcal{G}$, Proposition 1 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 $e_1 e_3$ 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. Table 3 shows the run using the optimization based on the enable set. Only the content of Δ is given for simplicity. At e_1, a monitor state is created for $\langle p_1 \rangle$ since it is the creation event. At e_2, no action is taken since $\mathsf{enable}_{\mathcal{G}}(e_2) = \{\emptyset\}$. 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 $\mathsf{enable}_{\mathcal{G}} e_3 = \{P_1\}$. 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	$\langle p_1 \rangle$: $\sigma(i, e_1)$	$\langle p_1 \rangle$: $\sigma(i, e_1)$	$\langle p_1 \rangle$: $\sigma(i, e_1)$ $\langle p_1, p_2 \rangle$: $\sigma(\sigma(i, e_1), e_3)$

Table 3: An unsound example of the enable set.

To avoid unsoundness, we introduce the notion of disable stamps of events. $\mathsf{disable} : [X \overset{\circ}{\rightarrow} V_X] \rightarrow \mathsf{integer}$ maps a parameter instance to a timestamp, represented as an integer. For parameter instance θ, $\mathsf{disable}(\theta)$ gives the time when last event with θ was received. We maintain timestamps for monitors using a mapping $\mathcal{T} : [[X \overset{\circ}{\rightarrow} V_X] \overset{\circ}{\rightarrow} \mathsf{integer}]$. $\mathcal{T}$ 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 $\mathcal{T}(\theta)$. Table 4 shows the evolution of disable and $\mathcal{T}$ when processing the trace in Table 3

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. $\theta'' \not\sqsubset \theta'$ 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 ($\theta'' \not\sqsubset \theta'$). 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 Table 4, 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	$\langle\rangle$: 1
disable	$\langle p_1\rangle$: 1	$\langle p_1\rangle$: 1 $\langle p_2\rangle$: 2	$\langle p_1\rangle$: 1 $\langle p_2\rangle$: 2 $\langle p_1, p_2\rangle$: 3

Table 4: Sound monitoring using 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. Table 5 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, although its enable set is not met, $\Delta(\langle P_2 \mapsto p_2\rangle)$ 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 $\{P_2\} \notin \mathsf{enable}_{\mathcal{G}}(e_1)$. 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 $\langle p_2\rangle \not\sqsubseteq \langle p_1\rangle$. In general, 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 $\mathcal{G}$. Intuitively, such 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 $\mathsf{Dom}(\theta'') \notin \mathsf{enable}_{\mathcal{G}}(e')$. Therefore, when we combine θ'' and θ' in θ, the trace slice for θ cannot belong to $\mathcal{G}$ according to the definition of the enable set.

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

Algorithm $\mathbb{D}\langle X\rangle(M = (S, \mathcal{E}, \mathcal{C}, 1, \sigma, \gamma))$
Input: mapping $\mathsf{enable}_{\mathcal{G}} : [\mathcal{E} \overset{\circ}{\to} \mathcal{P}_f(\mathcal{P}_f(X))]$
Globals: mapping $\Delta : [[X \overset{\circ}{\to} V_X] \overset{\circ}{\to} S]$
mapping $\mathcal{T} : [[X \overset{\circ}{\to} V_X] \overset{\circ}{\to} \text{integer}]$
mapping $\mathcal{U} : [X \overset{\circ}{\to} V_X] \to \mathcal{P}_f([X \overset{\circ}{\to} V_X])$
mapping disable : $[[X \overset{\circ}{\to} V_X] \overset{\circ}{\to} \text{integer}]$
integer timestamp
Initialization: $\mathcal{U}(\theta) \leftarrow \emptyset$ for any θ, timestamp $\leftarrow 0$

function main($e\langle\theta\rangle$)
1 if $\Delta(\theta)$ undefined then
2 ⋮ createNewMonitorState($e\langle\theta\rangle$)
3 ⋮ if $\Delta(\theta)$ 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 $\theta' \in \{\theta\} \cup \mathcal{U}(\theta)$ s.t. $\Delta(\theta')$ defined do
10 ⋮ $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e)$
11 endfor

function createNewMonitorStates($e\langle\theta\rangle$)
1 foreach $X_e \in \mathsf{enable}_{\mathcal{G}}(e)$ (in reversed topological order) do
2 ⋮ if $\mathsf{Dom}(\theta) \not\supseteq X_e$then
3 ⋮ ⋮ $\theta_m \leftarrow \theta'$ s.t. $\theta' \sqsubset \theta$ and $\mathsf{Dom}(\theta') = \mathsf{Dom}(\theta) \cap X_e$
4 ⋮ ⋮ foreach $\theta'' \in \mathcal{U}(\theta_m) \cup \{\theta_m\}$ s.t. $\mathsf{Dom}(\theta'') = X_e$ do
5 ⋮ ⋮ ⋮ if $\Delta(\theta'')$ defined and $\Delta(\theta'' \sqcup \theta)$ undefined then
6 ⋮ ⋮ ⋮ ⋮ defineTo($\theta'' \sqcup \theta, \theta''$)
7 ⋮ ⋮ ⋮ endif
8 ⋮ ⋮ endfor
9 ⋮ endif
10 endfor

function defineNew(θ)
1 foreach $\theta'' \sqsubset \theta$ do
2 ⋮ if $\Delta(\theta'')$ defined then return endif
3 endfor
4 $\Delta(\theta) \leftarrow 1$
5 $\mathcal{T}(\theta) \leftarrow$ timestamp
6 timestamp ← timestamp + 1
7 foreach $\theta'' \sqsubset \theta$ do
8 ⋮ $\mathcal{U}(\theta'') \leftarrow \mathcal{U}(\theta'') \cup \{\theta\}$
9 endfor

function defineTo(θ, θ')
1 foreach $\theta'' \sqsubseteq \theta$ s.t. $\theta'' \not\sqsubseteq \theta'$ do
2 ⋮ if $\mathsf{disable}(\theta'') > \mathcal{T}(\theta')$ or $\mathcal{T}(\theta'') < \mathcal{T}(\theta')$ then
3 ⋮ ⋮ return
4 ⋮ endif
5 endfor
6 $\Delta(\theta) \leftarrow \Delta(\theta')$
7 $\mathcal{T}(\theta) \leftarrow \mathcal{T}(\theta')$
8 foreach $\theta'' \sqsubset \theta$ do
9 ⋮ $\mathcal{U}(\theta'') \leftarrow \mathcal{U}(\theta'') \cup \{\theta\}$
10 endfor

Figure 3: Optimized monitoring algorithm $\mathbb{D}\langle X\rangle$

Event	$e_2\langle p_2\rangle$	$e_1\langle p_1\rangle$	$e_3\langle p_1, P_2 \mapsto p_2\rangle$
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 $\langle p_1\rangle$: 2	$\langle p_2\rangle$: 1 $\langle p_1\rangle$: 2
disable	$\langle p_2\rangle$: 1	$\langle p_2\rangle$: 1 $\langle p_1\rangle$: 2	$\langle\rangle$: 1 $\langle\rangle$: 2 $\langle p_1, p_2\rangle$: 3

Table 5: Another monitoring using timestamps.

Figure 3. This algorithm makes use of the data structures discussed above, namely, $\mathsf{enable}_{\mathcal{G}}$, 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 $\mathsf{enable}_{\mathcal{G}}(e)$ and looks for parameter instances whose domains are in $\mathsf{enable}_{\mathcal{G}}(e)$ 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 less lists in $\mathcal{U}$.

If e is a creation event then a monitor state for θ is initialized (lines 3 to 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 increased. 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 $\mathcal{T}(\theta')$.

3.3.1 Proofs of Correctness

We fix a trace $\tau = e_1e_2...e_n$, a Monitor $M = (S, \mathcal{E}, \mathcal{C}, 1, \sigma, \gamma)$ 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 the convenience of discussion, we also let timestamp : [integer $\overset{\circ}{\to}$ 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 $\mathbb{T}$ in $\mathbb{D}\langle X\rangle$ have the following properties:

Proposition 2 *The follow holds for timestamp and $\mathbb{T}$ 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. For 1., it is obviously since timestamp is monotone 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 : $[[X \overset{\circ}{\to} V_X] \overset{\circ}{\to} \mathsf{integer}]$ is defined as follows: $\mathsf{set}(\theta) = k$ if $\Delta(\theta)$ is initialized at e_k. Function MT : $[[X \overset{\circ}{\to} V_X] \overset{\circ}{\to} [X \overset{\circ}{\to} V_X]^*]$ is defined as follows: $\mathsf{MT}(\theta) = \theta_1...\theta_m$ where $\theta_m = \theta$, θ_1 is initialized with i, 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 3 *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 $\mathsf{MT}(\theta)$, $\theta_i \sqsubset \theta_j$ if $i < j$.*
2. *If $\mathsf{MT}_{\mathbb{D}}(\theta) = \theta_1...\theta_m$ then $\mathbb{T}(\theta) = \mathsf{timestamp}(\mathsf{set}_{\mathbb{D}}(\theta_1))$.*
3. *If $\mathsf{set}_{\mathbb{D}}(\theta)$ is defined then $\mathsf{set}_{\mathbb{C}}(\theta)$ is defined and $\mathsf{set}_{\mathbb{C}}(\theta) \leq \mathsf{set}_{\mathbb{D}}(\theta)$.*
4. *If $\mathsf{set}_{\mathbb{C}}(\theta) = \mathsf{set}_{\mathbb{D}}(\theta)$ 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 $\mathsf{set}_{\mathbb{C}}(\theta) = \mathsf{set}_{\mathbb{D}}(\theta)$ and $\mathsf{MT}_{\mathbb{C}}(\theta) = \mathsf{MT}_{\mathbb{D}}(\theta)$ 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, $\mathbb{T}(\theta) = \mathsf{timestamp}(k)$ 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 $m < j$, $\mathbb{T}(\theta'') = \mathsf{timestamp}(\mathsf{set}_{\mathbb{D}}(\theta_1))$. 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. $\mathbb{T}(\theta_j) = \mathbb{T}(\theta_{j-1})$ according to line 7 in defineTo in $\mathbb{D}\langle X\rangle$. By induction, $\mathbb{T}(\theta) = \mathbb{T}(\theta_{j-1}) = \mathsf{timestamp}(\mathsf{set}_{\mathbb{D}}(\theta_1))$.

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 afterwards 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 $\mathbb{D}\langle X\rangle$ perform equivalently on monitors that are created from the initial state.

Lemma 1 *The following hold for MT:*

1. *If $\mathsf{MT}_{\mathbb{C}}(\theta) = \theta$ then $\mathsf{MT}_{\mathbb{D}}(\theta) = \theta$ and $\mathsf{set}_{\mathbb{C}}(\theta) = \mathsf{set}_{\mathbb{D}}(\theta)$.*
2. *If $\mathsf{MT}_{\mathbb{D}}(\theta) = \theta$ then $\mathsf{MT}_{\mathbb{C}}(\theta) = \theta$ and $\mathsf{set}_{\mathbb{C}}(\theta) = \mathsf{set}_{\mathbb{D}}(\theta)$.*

Proof.
1. If $\mathsf{MT}_{\mathbb{C}}(\theta) = \theta$, suppose $\mathsf{set}_{\mathbb{C}}(\theta) = k$. 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 3 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 3 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 3 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 3 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 4 *For algorithms $\mathbb{C}^+\langle X\rangle$ and $\mathbb{D}\langle X\rangle$, the following hold:*

1. *If $\mathsf{MT}_{\mathbb{C}}(\theta) = \mathsf{MT}_{\mathbb{D}}(\theta)$ then for any $\theta' \in \mathsf{MT}_{\mathbb{C}}(\theta)$, $\mathsf{set}_{\mathbb{C}}(\theta') = \mathsf{set}_{\mathbb{D}}(\theta')$.*
2. *If $\mathsf{MT}_{\mathbb{C}}(\theta) = \mathsf{MT}_{\mathbb{D}}(\theta)$ 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 3.3. Let $\mathsf{set}_{\mathbb{C}}(\theta_j) = k$ and $e_k\langle\theta''\rangle$. Since $\theta'' \sqcup \theta_{j-1} = \theta_j$, we have $\theta'' \not\sqsubseteq \theta_{j-1}$. 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 3.5. □

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 5 *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 $\mathsf{set}_{\mathbb{C}}(\theta_i) = k$, we have $\mathsf{Dom}(\theta_{i-1}) \in \mathsf{enable}_{\mathcal{G}}(e_k)$.*
2. *If $\gamma(\Delta^{\tau}_{\mathbb{C}}(\theta)) \in \mathcal{G}$ then $\mathsf{MT}_{\mathbb{C}}(\theta) = \mathsf{MT}_{\mathbb{D}}(\theta)$.*

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 [16]. 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, $\mathsf{Dom}(\theta_{i-1}) \in \mathsf{enable}_{\mathcal{G}}(e_k)$ 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 1.1. 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 3 3., $\Delta_{\mathbb{D}}(\theta_j)$ is undefined before e_k. Also, $\theta' \sqcup \theta_{j-1} = \theta_j$ according to line 15 in main in $\mathbb{C}^+\langle X\rangle$.

By 1., $\mathsf{Dom}(\theta_{j-1}) \in \mathsf{enable}_{\mathcal{G}}(e_k)$. Hence, $\Delta_{\mathbb{D}}(\theta_j)$ will be defined at e_k because of the loop from line 4 to 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 [16]. 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 6 *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 3.2, $\Delta_{\mathbb{D}}(\theta^j_1)$ 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 in $\mathbb{D}\langle X\rangle$ 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 2.1. 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 3.3. 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 [16], 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 5 and Proposition 4.2, $\Delta^\tau_{\mathbb{D}}(\theta) = \Delta^\tau_{\mathbb{C}}(\theta)$. Hence, $\gamma(\Delta^\tau_{\mathbb{D}}(\theta)) = \gamma(\Delta^\tau_{\mathbb{C}}(\theta))$.

2. Similarly, it follows by Proposition 6 and Proposition 4.2.

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.

4 Implementation and Evaluation

We implemented code generation for Algorithm $\mathbb{D}\langle X\rangle$ in JavaMOP2, which generates AspectJ code from JavaMOP

	SafeMapIterator			SafeSyncCollection			SafeSyncMap			SafeIterator			SafeFile		SafeFileWriter	
	TM	MOP	MOP-O	TM	MOP	MOP-O	TM	MOP	MOP-O	TM	MOP	MOP-O	MOP	MOP-O	MOP	MOP-O
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

Table 6: Average percent runtime overhead for Tracematches(TM), JavaMOP 2.0 (MOP), and JavaMOP 2.0 with Enable Set Optimization(MOP-O) (convergence within 3%, OOM = Out of Memory).

specifications. The indexing technique proposed in [7] is used to implement all the mappings in the algorithm. Some optimizations were applied to achieve more efficient monitoring code. First, a function is generated for each event and in the function, $\mathsf{enable}_{\mathcal{G}}(e)$ is statically determined. So the main loop in createNewMonitorStates is unrolled at code generation time. 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 used as constants in code generation to remove unnecessary runtime computation. 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.

Another observation for optimization is that the inner loop (lines 4 to 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 $X_e \in \mathsf{enable}_{\mathcal{G}}(e)$. Specifically, given an event definition $\mathcal{D}_{\mathcal{E}}$, for any event e and every $X_e \in \mathsf{enable}_{\mathcal{G}}(e)$, 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 and no runtime checking is needed.

We evaluated JavaMOP2 on the DaCapo benchmark suite[2]. Tracematches and another version of JavaMOP2 which disables the enable set optimization are also evaluated for comparison. We omitted other runtime systems because they have been evaluated and compared with either Tracematches or JavaMOP in other papers [1, 7]. The centralized monitoring mode of JavaMOP was used in the evaluation. Also, we noticed that Soot [17], the underlying bytecode engine for Tracematches, cannot handle the Dacapo benchmark properly, resulting in less instrumentation points in the pmd program. Accordingly, we modify our specification to have the same scope of instrumentation for a fair comparison. All results can be found at [12].

Experimental Settings. Our experiments were performed on a machine with 1.5GB 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. It contains eleven open source programs [2]: antlr, bloat, chart, eclipse, fop, hsqldb, jython, luindex, lusearch, pmd, and xalan. 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 runtime converges. After this convergence, the runtime is stabilized within 3%, thus numbers in Table 6 should be interpreted as "$\pm$3%". Furthermore, additional code introduced by the AspectJ weaving process changes the program structure in DaCapo, and sometimes this causes the benchmark to run a little bit faster due to better concurrency interleaving and/or cache layout.

Properties. We used the following properties in our experiments. They were borrowed from [3, 4, 15].

- SafeMapIterator: 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.

SafeMapIterator, SafeSyncCollection, SafeMap and SafeFile could not be monitored in JavaMOP1, 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 overheads.

Results and Discussions. Table 6 summaries the results of our experiments. It shows the percent overheads of JavaMOP2 with the enable set optimization, JavaMOP2 without the enable set optimization and Tracematches. All the properties were heavily monitored in the experiments. As shown in [7], millions of parameter instances were observed for some properties under monitoring, e.g., SafeIterator, putting a critical test on the generated monitoring code. All three systems generated unnoticeable runtime overhead in most experiments, showing their efficiency. For the optimized JavaMOP2, only 7 out of 66 cases caused more than 10% runtime overhead. The numbers for the non-optimized JavaMOP2 and Tracematches are 9 out of 66 and 15 out of 44, respectively. Figure 4 shows the comparison among three systems using the cases where at least two of them generated more than 10% overhead. In all cases, the optimized JavaMOP2 outperformed the other two and the non-optimized JavaMOP2 is better than Tracematches and comparable with the optimized one in most cases. It thus shows that JavaMOP2 provides a more 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 the optimized JavaMOP2 is about 20% less than the non-optimized one. 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-SafeMapIterator and pmd-SafeMapIterator, where both the non-optimized JavaMOP2 and Tracematches crashed, the optimized JavaMOP2 managed to finish the executions with overehads that are reasonable for many applications, e.g., testing.

5 Conclusion

Efficient monitoring of parametric properties turns 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 and property-optimized semantics of parametric traces, overcomes these limitations, while being comparatively quite efficient, as our evaluation shows.

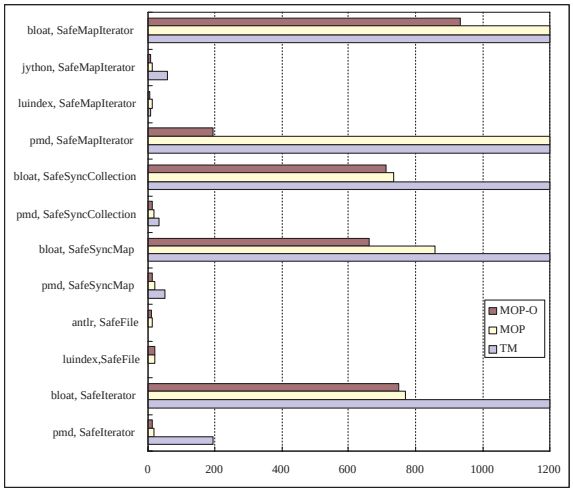

Figure 4: Comparison of optimized JavaMOP2, non-optimized JavaMOP2 and Tracematches

References

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

[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, and K. S. McKinley. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

[3] E. Bodden, F. Chen, and G. Roşu. Dependent advice: A general approach to optimizing history-based aspects (extended version). Technical Report abc-2008-2, Oxford University, 2008.

[4] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP*, 2007.

[5] F. Chen, D. Jin, P. Meredith, and G. Roşu. Efficient Formalism-Independent Monitoring of Parametric Properties (Extended Version). Technical Report UIUCDCS-R-2008-2977, University of Illinois at Urbana-Champaign, 2008.

[6] F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Runtime Verif.*, 2003.

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

[8] M. d'Amorim and K. Havelund. Event-based runtime verification of Java programs. *SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

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

[10] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *OOPSLA*, 2005.
[11] K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In *Runtime Verif.*, 2001.
[12] Javamop results page. http://fsl.cs.uiuc.edu/index.php/JavaMOP_ICSE_Results.
[13] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a runtime assurance tool for Java. In *Runtime Verif.*, 2001.
[14] M. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA*, 2005.
[15] P. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. In *ASE '08*, 2008.
[16] G. Roşu and F. Chen. Parametric Trace Slicing and Monitoring. Technical Report UIUCDCS-R-2008-2977, University of Illinois at Urbana-Champaign, 2008.
[17] Soot website. http://www.sable.mcgill.ca/soot/.