Efficient Formalism-Independent Monitoring of Parametric Properties

Feng Chen  Patrick O’Neil Meredith  Dongyun Jin  Grigore Roșu
University of Illinois at Urbana-Champaign
\{fengchen, pmeredit, djin3, grosu\}@cs.uiuc.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, a thorough evaluation shows that our technique outperforms other state-of-art techniques optimized for particular logics or properties.1

I. 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 [1], [2], [3], [4], [5], [6], [7], [8], [9], 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:

\[
\forall m, c, i. \big( create_coll(m, c) \land (create_iterator(c, i) \land (update_map(m) \land use_iterator(i))) \big)
\]

Where \(create_coll\) is creating a collection from a map, \(create_iterator\) is creating an iterator from a collection, \(update_map\) is updating the map, and \(use_iterator\) is using the iterator; \(\land\) 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. The violating behavior can also be specified as an extended regular expression (ERE) that is more understandable for programmers but less concise:

\[
\forall m, c, i. \big( create_coll(m, c) \land use_iterator(i) \land update_map(m) \land create_iterator(c, i) \land \big)\]

\[
use_iterator \land update_map(m) + use_iterator(i)
\]

In fact, as shown in [10], an extended version of this paper that shows the UnsafeMapIterator example as it appears in JavaMOP, many different logics can be used to describe this property, depending on the user’s preference. This observation emphasizes our belief that no silver-bullet logic exists for all application domains, making a formalism-independent support for parametric properties desirable in practice.

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 \(update_map(m)\) is received, we need to find all \(create_coll(m, c)\) events with the same binding for \(m\), and transitively, all \(create_iterator(c, i)\) with the same \(i\) as that \(create_coll\).

Several approaches were introduced to support the monitoring of parametric specifications, including Eagle [4], Tracematches [5], [6], PQL [7], PTQL [8] and MOP [9]. However,

---

1Supported 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.
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 [9] and Section VII 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 [9] 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. This technique is based on the theoretical results in [11], which was focused on a general, theoretical solution for handling parametric traces and proposed a conceptual algorithm\(^2\).

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 V.

Our technique has been implemented in the latest version of JavaMOP\(^3\). 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 [11], 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. For the non-optimized JavaMOP and Tracematches, 9 out of 66 and 15 out of 44 cases, respectively, have more than 10% runtime overhead. 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. 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.

3) An extensive evaluation and comparison of the proposed solution with Tracematches, an efficient monitoring system for regular pattern properties [5], [6].

### II. Approach Overview

We next use the UnsafeMapIterator example from Section I to illustrate our technique. Consider the example trace of eleven events in Fig. 1 over the events from UnsafeMapIterator. 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., \(<m_1,c_1>\) that binds parameters \(m\) and \(c\) with a map object \(m_1\) and a collection \(c_1\), respectively. A trace with parametrized events is called a parametric trace.

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 III, 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 \(<m_1,c_2>\) is less informative than \(<m_1,c_2,i_3>\) because the parameters they both bind, \(m\) and \(c\),

<table>
<thead>
<tr>
<th>#</th>
<th>Event</th>
<th>#</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>create_coll((m_1,c_1))</td>
<td>7</td>
<td>update_map((m_1))</td>
</tr>
<tr>
<td>2</td>
<td>create_coll((m_1,c_2))</td>
<td>8</td>
<td>use_iter((i_2))</td>
</tr>
<tr>
<td>3</td>
<td>create_iter((c_1,i_1))</td>
<td>9</td>
<td>create_coll((m_2,c_3))</td>
</tr>
<tr>
<td>4</td>
<td>create_iter((c_1,i_2))</td>
<td>10</td>
<td>create_iter((c_3,i_4))</td>
</tr>
<tr>
<td>5</td>
<td>use_iter((i_1))</td>
<td>11</td>
<td>use_iter((i_4))</td>
</tr>
<tr>
<td>6</td>
<td>create_iter((c_2,i_3))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 1. Possible Execution Trace Over for UnsafeMapIterator.**

\(^2\)Note that the evaluation results in [11] are based on the technique presented in this paper. The technique was only very briefly mentioned in [11], due to its different focus, and because the optimization had not yet been formalized.

\(^3\)JavaMOP is the Java specialization of MOP, which is itself a framework generic in specification formalisms [9].
agree on their values, \(m_1\) and \(c_2\), respectively, and \(\langle m_1, c_2, i_3 \rangle\) binds one more parameter. Fig. 2 shows the trace slices and their corresponding parameter bindings contained in the trace in Fig. 1. The Status column denotes the output category that the slice falls into (for ERE). In 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. Fig. 3 shows the set of monitors created for the trace in Fig. 1, 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 push-down 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 Fig. 1, 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 Fig. 1, 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 III-C.

An algorithm to build parameter instances from observed events, like the one introduced in [11], often creates many useless monitor instances leading to prohibitive runtime overhead. For example, Fig. 2 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) it reduces the needed memory space, and (2), more importantly, monitoring efficiency is improved because 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 IV. 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 may 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, while 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 number of needed monitors using static program analysis, e.g., the one introduced in [12]. However, such techniques are based on the program targeted for monitoring, leading to two drawbacks: (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\(^4\), 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 different programs, because the enable set is derived only from the property.

\(^4\)The analysis is bounded above 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.
III. PARAMETRIC MONITORING

In this section, we briefly introduce the core semantics of parametric monitoring based on parametric trace slicing to make this paper self-contained. More details, including further formal definitions and proofs, can be found in [11].

A. Events, Traces and Properties

Definition 1: Let $E$ be a set of (non-parametric) events, called base events or simply events. An $E$-trace, or simply a (non-parametric) trace when $E$ is understood, is any finite sequence of events in $E$, that is, an element in $E^*$. If event $e \in E$ appears in trace $w \in E^*$ then we write $e \in w$.

For example, $\{\text{create\_coll, create\_iter, use\_iter, update\_map}\}$ is the set of events in the UnsafeMapIterator property in Section I; $\text{create\_coll create\_iter use\_iter update\_map}$ is a trace.

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

Intuitively, properties are trace classifiers, that is, mappings partitioning the space of traces into categories (violating traces, validating traces, *?” or undecided traces, etc.).

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 $E$ is a set of events (Definition 1), then let $E(X)$ denote the set of corresponding parametric events $e(\theta)$, where $e$ is a base event in $E$ and $\theta$ is a parameter instance, i.e., an element in $[X \overset{\circ}{\rightarrow} V]$, the set of partial maps from $X$ to $V$. $\perp$ is the empty partial map. A parametric trace is a trace with events in $E(X)$, i.e., a word in $E(X)^*$.

For example, if $X = \{m, c, i\}$ is a set of parameters (of types $\{\text{Map, Collection, Iterator}\}$, respectively) and $V = \{m_1, c_1, i_1, i_2\}$, then $\text{create\_coll}(m \mapsto m_1, c \mapsto c_1)$, $\text{create\_iter}(c \mapsto c_1, i \mapsto i_1)$, and $\text{use\_iter}(i \mapsto i_1)$, are parametric events and $\text{create\_coll}(m \mapsto m_1, c \mapsto c_1) \text{ create\_iter}(c \mapsto c_1, i \mapsto i_1) \text{ use\_iter}(i \mapsto i_1)$ is a parametric trace. In this paper we simplify the representation of parametric instances by hiding their domains when they are understood from the context, e.g., $\text{create\_coll}(m_1, c_1)$ instead of $\text{create\_coll}(m \mapsto m_1, c \mapsto c_1)$, and $(m_1, c_1)$ instead of $(m \mapsto m_1, c \mapsto c_1)$.

Definition 4: Parameter instance $\theta$ is compatible with parameter instance $\theta'$ if for any parameter $x \in \text{Dom}(\theta) \cap \text{Dom}(\theta')$, $\theta(x) = \theta'(x)$. We can combine compatible parameter instances $\theta$ and $\theta'$, written $\theta \sqcup \theta'$, as follows:

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

$\theta'$ is less informative than $\theta$, 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, $(m_1, c_1)$ and $(c_1, i_1)$ are compatible and $(m_1, c_1) \sqcup (c_1, i_1) = (m_1, c_1, i_1)$. Logically, $\perp$ is compatible with, and less informative than, all parameter instances, because it does not bind any parameters.

Definition 5: Given parametric trace $\tau \in E(X)^*$ and $\theta$ in $[X \overset{\circ}{\rightarrow} V]$, we let the $\theta$-trace slice $\tau_\theta \in E^*$ be the non-parametric trace in $E^*$ defined as follows:

- $\epsilon|_\theta = \epsilon$, where $\epsilon$ is the empty trace/word, and
- $\{\tau \in E^* \mid \tau|_\theta \in E^* \text{ and } \tau|_\theta \subseteq \theta\}$

Therefore, the trace slice $\tau_\theta$ first filters out all the parametric events that are not relevant for the instance $\theta$, i.e., which contain instances of parameters that $\theta$ does not care about, and then, for the remaining events relevant to $\theta$, it forgets the parameters so that the trace can be checked against base, non-parametric properties. Consider the parametric trace $\text{create\_coll}(m_1, c_1)$ $\text{create\_iter}(c_1, i_1)$ $\text{use\_iter}(i_1)$ $\text{update\_map}(m_1)$ $\text{create\_coll}(m_1, c_2)$. The trace slice for $(m_1)$ is $\text{update\_map}$, for $(m_1, c_2)$ is $\text{create\_coll}$, and for $(m_1, c_1, i_1)$ is $\text{create\_coll create\_iter use\_iter update\_map}$.

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

$$\Lambda X.P : E(X)^* \rightarrow [X \overset{\circ}{\rightarrow} V] \rightarrow C$$

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

$\Lambda X.P$ 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 $\theta$ 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 [5], [6], 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 $\text{start(resource)} \text{use(client,resource)} *$. The trace $\text{start(resource1)}$ should match the pattern, but it would not be matched in Tracematches, because there is no instantiation of the parameter client.

B. 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 7: A monitor $M$ is a tuple $(S, E, C, 1, \sigma : S \times E \rightarrow S, \gamma : S \rightarrow C)$, where $S$ is a set of states, $E$ is a set of input events, $C$ is a set of output categories, $1 \in S$ is the initial state, $\sigma$ is the transition function, and $\gamma$ is the output function. The transition function is extended to $\sigma : S \times E^* \rightarrow S$ as expected: $\sigma(s, e) = s$ and $\sigma(s, we) = \sigma(\sigma(s, w), e)$ for any $s \in S$, $e \in E$, and $w \in E^*$.

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 [13] 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. The following is natural:

Definition 8: \( M = (S, E, C, 1, \sigma, \gamma) \) is a monitor for property \( P : E^* \rightarrow C \) iff \( \forall w \in E^*, \gamma(\sigma(1, w)) = P(w) \).

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 9: Given parameters \( X \) with corresponding values \( V \) and \( M = (S, E, C, 1, \sigma : S \times E \rightarrow S, \gamma : S \rightarrow C) \), we define the parametric monitor \( \Lambda X.M \) as the monitor

\[
([X \stackrel{\circ}{\rightarrow} V] \rightarrow S), E(X), ([X \stackrel{\circ}{\rightarrow} V] \rightarrow C), \lambda \theta \lambda, \Lambda X.\sigma, \Lambda X.\gamma,
\]

with \( \Lambda X.\sigma \): \( ([X \stackrel{\circ}{\rightarrow} V] \rightarrow S) \times E(X) \rightarrow ([X \stackrel{\circ}{\rightarrow} V] \rightarrow S) \) and \( \Lambda X.\gamma \): \( ([X \stackrel{\circ}{\rightarrow} V] \rightarrow S) \rightarrow ([X \stackrel{\circ}{\rightarrow} V] \rightarrow C) \) defined as follows

\[
(\Lambda \theta \lambda \lambda \theta)(\delta, e(\theta'))(\theta') = \begin{cases} 
\sigma(\delta(\theta'), e) & \text{if } \theta' \subseteq \theta \\
\delta(\theta') & \text{if } \theta' \not\subseteq \theta 
\end{cases}
\]

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

For the sake of rigor, Definition 9 may seem very complicated. All it says, however, is that a state \( \delta \) of parametric monitor \( \Lambda X.M \) maintains a state \( \delta(\theta) \) of \( M \) for each parameter instance \( \theta \), takes parametric events as input, and outputs categories indexed by parameter instances (one output category of \( M \) per parameter instance). This is analogous to the intuitive example in Fig. 2, 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 [11])

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 straightforwardly extending it as per Definition 9.

C. Monitoring with Creation Events

Fig. 4 shows the algorithm \( \mathbb{C}^+(X) \) 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 slightly extends algorithm \( \mathbb{C}(X) \) in [11] to support creation events. \( \mathbb{C}^+(X) \) is justified and motivated by experience with implementing and evaluating \( \mathbb{C}(X) \) in [11], mainly by the following observation: one often chooses to start monitoring at the witness of a specific set of events (rather than the beginning of the program). For example, when we monitor the UnsafeMapIterator property in Section I, 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 events before the creation. We claim such events that lead to creation of new monitor states (monitor) creation events. Obviously, \( \mathbb{C}(X) \) can be regarded as a special case of \( \mathbb{C}^+(X) \), when all the events are creation events. The proof of \( \mathbb{C}^+(X) \) is easily derived from the proof of \( \mathbb{C}(X) \) in [11].

The first challenge to online monitoring of a parametric property is that the state space of potential parameter instances is infinite. We encode partial functions \( ([X \stackrel{\circ}{\rightarrow} V] \rightarrow Y) \), which map some parameter instances \( [X \stackrel{\circ}{\rightarrow} V] \) to elements in \( Y \), as tables with entries indexed by parameter instances in \( [X \stackrel{\circ}{\rightarrow} V] \) 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. Two mappings are used in \( \mathbb{C}^+(X) \): \( \Delta \) and \( U. \) \( \Delta \) stores the monitor states for parameter instances, and \( U \) maps a parameter instance \( \theta \) to all the parameter instances that have been defined and are properly
Therefore, a new monitor instance is created for the combined formalism. However, this generality leads to extra novel optimization based on the concept of enable sets. To motivate the optimization, let us continue the run in Fig. 5 to process one more event, use_iter(i1). The result is shown in Fig. 6. use_iter(i1) is not a creation event and no monitor instance is created for (i1). Since (i1) is compatible with (m2, c2), a new monitor instance is defined for (m2, c2, i1). The monitor instance for (m1, c1, i1) is then updated according to use_iter because (i1) is less informative than (m1, c1, i1). U is also updated to add (m2, c2, i1) to the lists for all the parameter instances less informative than (m2, c2, i1). New entries are added into U during the update since some of less informative parameter instances, e.g., (m2, i1), have not been used before this event.

### IV. Beyond C+(X): Enable Sets

C+(X) 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 new optimization based on the concept of enable sets.

<table>
<thead>
<tr>
<th>Event</th>
<th>update_map(m1)</th>
<th>create_coll(m1, c1)</th>
<th>create_coll(m2, c2)</th>
<th>create_iter(c1, i1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>∆</td>
<td>∅</td>
<td>⟨m1, c1⟩:σ(i, create_coll)</td>
<td>⟨m2, c2⟩:σ(i, create_coll)</td>
<td>⟨m1, c1, i1⟩:σ(σ(i, create_coll), create_iter)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>⟨m1, c1⟩:σ(i, create_coll)</td>
<td>⟨m2, c2⟩:σ(i, create_coll)</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>∅</td>
<td>⟨m1⟩:⟨m1, c1⟩</td>
<td>⟨m2⟩:⟨m2, c2⟩</td>
<td>⟨m1, c1, i1⟩:⟨m1, c1, i1⟩</td>
</tr>
<tr>
<td></td>
<td></td>
<td>⟨m1⟩:⟨m1, c1⟩</td>
<td>⟨m2⟩:⟨m2, c2⟩</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>⟨m1⟩:⟨m1, c1⟩</td>
<td>⟨m2⟩:⟨m2, c2⟩</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>⟨m1⟩:⟨m1, c1⟩</td>
<td>⟨m2⟩:⟨m2, c2⟩</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 5. Sample Run of C+(X). The first row gives the received events; the second and the third rows give the content of ∆ and U, respectively, after every event is processed. Monitor states are represented symbolically in the table, e.g., σ(i, create_coll) represents the state after the event create_coll.

<table>
<thead>
<tr>
<th>Event</th>
<th>use_iter(i1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>∆</td>
<td>⟨m1, c1⟩:σ(i, create_coll)</td>
</tr>
<tr>
<td></td>
<td>⟨m2, c2⟩:σ(i, create_coll)</td>
</tr>
<tr>
<td></td>
<td>⟨m1, c1, i1⟩:σ(σ(i, create_coll), create_iter, use_iter)</td>
</tr>
<tr>
<td></td>
<td>⟨m2, c2, i1⟩:σ(i, create_coll, use_iter)</td>
</tr>
<tr>
<td>U</td>
<td>⟨m1, c1⟩, ⟨m2, c2⟩, ⟨m2, c2, i1⟩, ⟨m1, c1, i1⟩</td>
</tr>
<tr>
<td></td>
<td>⟨m1⟩:⟨m1, c1⟩, ⟨i1⟩:⟨m1, c1, i1⟩</td>
</tr>
<tr>
<td></td>
<td>⟨m2⟩:⟨m2, c2⟩, ⟨m2, c2, i1⟩</td>
</tr>
<tr>
<td></td>
<td>⟨m2⟩:⟨m2, c2⟩, ⟨m2, c2, i1⟩</td>
</tr>
<tr>
<td></td>
<td>⟨m2⟩:⟨m2, c2⟩, ⟨m2, c2, i1⟩</td>
</tr>
<tr>
<td></td>
<td>⟨m2⟩:⟨m2, c2⟩, ⟨m2, c2, i1⟩</td>
</tr>
<tr>
<td></td>
<td>⟨m2⟩:⟨m2, c2⟩, ⟨m2, c2, i1⟩</td>
</tr>
<tr>
<td></td>
<td>⟨m2⟩:⟨m2, c2⟩, ⟨m2, c2, i1⟩</td>
</tr>
<tr>
<td></td>
<td>⟨m2⟩:⟨m2, c2⟩, ⟨m2, c2, i1⟩</td>
</tr>
<tr>
<td></td>
<td>⟨m2⟩:⟨m2, c2⟩, ⟨m2, c2, i1⟩</td>
</tr>
</tbody>
</table>

Fig. 6. Following the Run of Fig. 5.
an iterator can be associated to only one collection. Hence, the monitor for \(m_2, c_2, i_1\) 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, \(C\) in Definition 2, is checked. For example, in the UnsafeMapIterator property in Section I, 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 check, before an execution, to infer automatically. Below, we show a simpler yet effective solution to avoid unnecessary creation of the specified pattern can be reached by the trace slice for \(m_2, c_2, i_1\), that is to say, the monitor for \(m_2, c_2, i_1\) 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 \(\Delta\) 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. How this information is used is discussed in Section VI.

Definition 10: Given \(\tau \in \mathcal{E}^*\) and \(e, e' \in \tau\), we denote that \(e'\) occurs before an occurrence of \(e\) in \(\tau\) as \(e' \prec \tau e\). Let the trace enable set of \(e \in \mathcal{E}\) be the function \(\text{enable}_{e,\tau} : \mathcal{E} \rightarrow \mathcal{P}_{f}(\mathcal{E})\), defined as: 

\[
\text{enable}_{e,\tau}(e) = \{e' \mid e' \prec \tau e\}.
\]

Note that if \(e \notin \tau\) then \(\text{enable}_{e,\tau}(e) = \emptyset\). 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 \(e' \in \text{enable}_{e,\tau}(e)\) has not been observed, then the (incomplete) execution being monitored will not produce the trace \(\tau\) 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 categories are called the goal of the monitoring.

Definition 11: Given \(P : \mathcal{E}^* \rightarrow \mathcal{C}\) and a set of categories \(\mathcal{G} \subseteq \mathcal{C}\) as the goal, the property enable set is defined as a function \(\text{enable}_{P,\mathcal{G}} : \mathcal{E} \rightarrow \mathcal{P}_{f}(\mathcal{P}_{f}(\mathcal{E}))\) with \(\text{enable}_{P,\mathcal{G}}(e) = \{\text{enable}_{e,\tau}(e) \mid P(\tau) \in \mathcal{G}\}\).

Intuitively, if event \(e\) is encountered during monitoring but none of event sets \(\text{enable}_{P,\mathcal{G}}(e)\) has been completely observed, the (incomplete) execution being monitored will not produce a trace \(\tau\) s.t. \(P(\tau) \in \mathcal{G}\). For example, given the regular expression specifying the UnsafeMapIterator property in Section I, where \(\mathcal{G}\) contains only the match, violation, and \(?\) categories, the second column in Fig. 7 shows the property enable sets of events in 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 \(\text{create_coll} \notin \text{enable}_{P,\mathcal{G}}(\text{use_iter})\). 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 create_coll use_iter from before. At the second event, use_iter, a new parameter instance can be constructed, namely, \(m_1, c_1, i_1\), and a monitor state \(s\) will be created for \(m_1, c_1, i_1\) if algorithm \(\mathcal{C}^+(\mathcal{X})\) is applied. However, since the trace slice for \(s\) is create_coll use_iter, we immediately know that \(s\) cannot reach state match. So there is no need to create and maintain \(s\) during monitoring if match is the goal.

A direct application of the above idea to optimize \(\mathcal{C}^+(\mathcal{X})\) 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 12: Given a property \(P : \mathcal{E}^* \rightarrow \mathcal{C}\), a set of categories \(\mathcal{G} \subseteq \mathcal{C}\) as the goal, a set of parameters \(X\) and a function \(\mathcal{D}_X : \mathcal{E} \rightarrow \mathcal{P}_{f}(\mathcal{P}(\mathcal{X}))\) mapping an event to its parameters, the property parameter enable set of event \(e \in \mathcal{E}\) is defined as a function \(\text{enable}_{P,\mathcal{G}, X}(e) : \mathcal{E} \rightarrow \mathcal{P}_{f}(\mathcal{P}(\mathcal{X}))\) as follows:

\[
\text{enable}_{P,\mathcal{G}, X}(e) = \{D_X(e') \mid e' \in \text{enable}_{e,\tau}(e) \mid P(\tau) \in \mathcal{G}\}.
\]

From now on, we use “enable set” to refer to “property parameter enable set” for simplicity. For example, given the regular pattern for the UnsafeMapIterator property in Section I and \(\mathcal{G} = \{\text{validating}\}\); the third column in Fig. 7 shows the parameter enable sets of events in UnsafeMapIterator. Then, given again the trace \(\text{create_coll}(m_1, c_1)\) use_iter\(i_1\), no monitor state need be created at the second event for \(m_1, c_1, i_1\)
since the parameter instance used to initialize the new monitor state, namely, \((m_1, c_1)\), is not in \(\text{enable}_G^X(\text{use}_{\text{iter}})\). 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. This allows the plugins to remain totally parameter agnostic. The following result guarantees the correctness of this approach:

**Proposition 2:** When algorithm \(\mathcal{C}^+(X)\) receives event \(e(\theta)\), if we use \(\theta'\) to define \(\theta \sqcup \theta'\) and \(\text{Dom}(\theta') \notin \text{enable}_G^X(e)\), then \(\gamma((\theta \sqcup \theta')) \notin G\) during the whole monitoring process.

### V. Computing Enable Sets

As we mentioned, the definition of the enable set is general and does not depend on a specific formalism to write the property. We next show two algorithms to compute enable sets for finite-state machine (FSM) based monitors and context-free grammars (CFG), respectively.

**Algorithm** \(\mathcal{E}N_{\text{fsm}}(FSM = (E, S, s_0, \delta))\)

**Globals:** mapping \(\mathcal{V}_\mu : S \rightarrow \mathcal{P}_f(\mathcal{P}_f(E))\)  
mapping \(\text{enable}_G^E : E \rightarrow \mathcal{P}_f(\mathcal{P}_f(E))\)

**Initialization:** \(\mathcal{V}_\mu(s) \leftarrow \emptyset\) for any \(s \in S\)  
\(\text{enable}_G^E(e) \leftarrow \emptyset\) for any \(e \in E\)

**function main()**
1. \(\text{compute}_{\text{enables}}(s_0, \emptyset)\)

**function compute_{\text{enables}}(s, \mu)\)
1. foreach defined \(\delta(s, e)\) do
2. : \(\text{enable}_G^E(e) \leftarrow \text{enable}_G^E(e) \cup \{\mu\}\)
3. : let \(\mu' \leftarrow \mu \cup \{e\}\)
4. : if \(\mu' \notin \mathcal{V}_\mu(s)\)
5. : \(\mathcal{V}_\mu(s) \leftarrow \mathcal{V}_\mu(s) \cup \{\mu'\}\)
6. : \(\text{compute}_{\text{enables}}(\delta(s, e), \mu')\)
7. : endfor
8. endfor

Fig. 8. FSM \(\text{enable}_G^E\) Computation Algorithm.

**Case 1: FSM** The algorithm in Fig. 8 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, FTLTL, and PTLTL (past time linear temporal logic). The algorithm assumes a finite state machine, defined as \(FSM = (E, S, s_0 \in S, \delta : S \times E \rightarrow S)\). \(E\) is the alphabet, traditionally listed as \(\Sigma\) but changed for consistency, because the alphabets of our FSMs are event sets. \(s_0\) is the start state, corresponding to \(1\) in the definition of a monitor. \(\delta\) is the transition partial function, taking a state and an event and potentially mapping to a next state for the machine. We assume that all states not reachable from the initial state and not coreachable from the states of interest (states of interest being those states \(s\) such that \(\gamma(s) \in G\)) are pruned from the FSM before running the algorithm, leaving the transitions that pointed to them undefined. \(\mathcal{V}_\mu\) is a mapping from states to sets of events; it is used to check for algorithm termination. \(\text{enable}_G^E\) is the output property enable set, which is converted into a parameter enable set by JavaMOP.

**Case 2: CFG** We also provide an algorithm to compute the \(\text{match}\) enable set for a context-free pattern, which has an infinite monitor state space, as briefly explained in what follows. This is a modification of the algorithm in Fig. 8.

Let \(G = \{\text{match}\}\). For \(\text{enable}_G^E\) and a given CFG \(G = (NT, E, P, S)\) we begin with all productions \(S \rightarrow \gamma\) and the set \(\mu_0 = \emptyset \in \mathcal{P}_f(E)\). For each production, we investigate each \(s \in \gamma\) (where \(e\) is, by abuse of notation, used to denote a symbol in a right hand side) from left to right. If \(s \in E\) we add \(\mu_i\) to \(\text{enable}_G^E(s)\), thus if \(s\) is the first symbol in \(\gamma\) we add \(\mu_0\). We then add \(s\) to \(\mu_i\), forming \(\mu_{i+1}\). If \(s \in NT\) we recursively invoke the algorithm, but rather than use \(\mu_0\), we use \(\mu_i\), 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 of the algorithm.

**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

5We assume a certain familiarity with context free patterns; definitions can be found in [13], together with explanations on CFG monitoring.
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 [12], 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 Fig. 6. 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 \( m_2, c_2 \) and \( i_1 \) 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 [12], 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, \( i_1 \) and \( m_1, c_1 \) need to be combined to build a new parameter instance because \( c_1 \) and \( i_1 \) are connected in the second event, \text{create} \( \text{coll}(m_1, c_1) \), in Fig. 6, but \( i_1 \) and \( m_2, c_2 \) 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.

VI. MONITORING WITH ENABLE SETS

We next integrate the concept of enable sets with algorithm \( \mathcal{C}^+(X) \), to improve performance and memory usage. All related proofs can be found in [10].

Given a set of desired value categories \( G \), Proposition 2 guarantees that we can optimize the monitoring process by omitting creating monitor states for certain parameter instances when an event is received using the enable set without missing any trace belonging to \( G \). However, skipping the creation of monitor states may result in false alarms, i.e., a trace that is not in \( G \) can be reported to belong to \( G \). Let us consider the following example. We monitor to find matching of a regular pattern \( e_1 e_3 \). Relevant events and their parameters are \( e_1(P_1), e_2(P_2), e_3(P_1, P_2) \). The observed trace is \( e_1(p_1) e_2(p_2) e_3(p_1, p_2) \). Also, suppose \( e_1 \) is the only creation event. Obviously, the trace does not match the pattern. Fig. 9 shows the run using the enable set optimization (i.e., not creating monitor states for parameter instances disallowed by the enable sets). Only the content of \( \Delta \) 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 \( \text{enable}^X_\bot (e_2) = \emptyset \). At \( e_3 \), a monitor state will be created for \( \langle p_1, p_2 \rangle \) using the monitor state for \( \langle P_1 \rightarrow p_1 \rangle \) since \( \text{enable}^X_\bot e_3 = \{ P_1 \} \). This way, \( e_2 \) is forgotten and a match of the pattern is reported incorrectly.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
Event & \( e_1(p_1) \) & \( e_2(p_2) \) & \( e_3(p_1, p_2) \) \\
\hline
\( \Delta \) & \( \langle p_1 \rangle; \sigma(i, e_1) \) & \( \langle p_1 \rangle; \sigma(i, e_1) \) & \( \langle p_1 \rangle; \sigma(i, e_1) \) \\
\hline
\( T \) & \( \langle p_1 \rangle; 1 \) & \( \langle p_1 \rangle; 1 \) & \( \langle p_1 \rangle; 1 \) \\
\hline
disable & \( \langle p_1 \rangle; 2 \) & \( \langle p_2 \rangle; 3 \) & \( \langle p_1, p_2 \rangle; 4 \) \\
\hline
\end{tabular}
\caption{Unsound Usage of the Enable Set.}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
Event & \( e_1(p_1) \) & \( e_2(p_2) \) & \( e_3(p_1, p_2) \) \\
\hline
\( \Delta \) & \( \langle p_1 \rangle; \sigma(i, e_1) \) & \( \langle p_1 \rangle; \sigma(i, e_1) \) & \( \langle p_1 \rangle; \sigma(i, e_1) \) \\
\hline
\( T \) & \( \langle p_1 \rangle; 1 \) & \( \langle p_1 \rangle; 1 \) & \( \langle p_1 \rangle; 1 \) \\
\hline
disable & \( \langle p_1 \rangle; 2 \) & \( \langle p_2 \rangle; 3 \) & \( \langle p_1, p_2 \rangle; 4 \) \\
\hline
\end{tabular}
\caption{Sound Monitoring Using Timestamps.}
\end{table}

A. Timestamping Monitors: Algorithm \( \mathbb{D}(X) \)

To avoid unsoundness, we introduce the notion of disable stamps of events. \text{disable} : [(\mathbb{X} \times \mathbb{V}) \rightarrow \text{integer}] maps a parameter instance to an integer timestamp. \text{disable}(\theta) gives the time when the last event with \( \theta \) was received. We maintain timestamps for monitors using a mapping \( T : [(\mathbb{X} \times \mathbb{V}) \rightarrow \text{integer}] \). \( T \) 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 \( \theta \) is created using the initial state when a creation event is received (i.e., using the defineNew function in algorithm \( \mathcal{C}^+(X) \)), \( T(\theta) \) is set to the time of creation; if a monitor state for \( \theta \) is created from the monitor state for \( \theta' \), \( T(\theta') \) is passed to \( T(\theta) \). Fig. 10 shows the evolution of \text{disable} and \( T \) while processing the trace in Fig. 9.

\( \text{disable} \) and \( T \) can be used together to track “skipped events”: when a monitor state for \( \theta \) is created using the monitor state for \( \theta' \), there exists some \( \theta'' \sqsubseteq \theta \) s.t. \( \theta'' \not\sqsubseteq \theta' \) and \text{disable}(\theta'') > \( T(\theta') \) then the trace slice for \( \theta \) does not belong to the desired value categories \( G \). Intuitively, \text{disable}(\theta'') > \( T(\theta') \) implies that an event \( e(\theta'') \) has been encountered after the monitor state for \( \theta' \) was created. But \( \theta'' \) 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 \( \theta \) does not belong to \( G \).
according to the definition of the enable set. Therefore, in Fig. 10, no monitor instance is created for \((p_1, p_2)\) at \(e_3\) because \(\text{disable}(p_2) > T(p_1)\).

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. If the skipped event is not a creation event, it does not affect the soundness of the algorithm because of the definition of creation events. In the above example, if the observed trace is \(e_2(p_2)e_1(p_1)e_3(p_1, p_2)\), we will ignore \(e_2\) and report the matching at \(e_3\) since \(e_1\) is the only creation event. It is more sophisticated (but not much different) when the skipped event is a creation event. The interested reader is referred [10] for more discussion.

Based on the above discussion, we develop a new parametric monitoring algorithm that optimizes algorithm \(C^+(X)\) using the enable set and timestamps, as shown in Fig. 11. This algorithm makes use of the mappings discussed above, namely, \(\text{enable}_\theta^X\), \(\Delta\), \(U\), \(\text{disable}\) and \(T\), and maintains an integer variable to track the timestamp. Similar to algorithm \(C^+(X)\), when event \(e(\theta)\) is received, algorithm \(D(X)\) 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 \(\text{createNewMonitorStates}\) in algorithm \(D(X)\). Unlike in algorithm \(C^+(X)\), where all the parameter instances less informative than \(\theta\) are searched to find all the compatible parameter instances using \(U\), \(\text{createNewMonitorStates}\) enumerates parameter sets in \(\text{enable}_\theta^X(e)\) and looks for parameter instances whose domains are in \(\text{enable}_\theta^X(e)\) and which are compatible with \(\theta\), also using \(U\). The inclusion check at line 2 in \(\text{createNewMonitorStates}\) is to omit unnecessary search since if \(\text{Dom}(\theta) \subseteq X_e\) then no new parameter instance will be created from \(\theta\). This way, \(\text{createNewMonitorStates}\) creates all the parameter instances from \(\theta\) whenever the enable set of \(e\) is satisfied using fewer lists in \(U\).

If \(e\) is a creation event then a monitor state for \(\theta\) is initialized (lines 3 - 5 in main). Note that \(\Delta(\theta)\) can be defined in function \(\text{createNewMonitorStates}\) if \(\Delta(\theta')\) has been defined for some \(\theta' \subsetneq \theta\). disable(\(e\)) is set to the current timestamp after all the creations and the timestamp is increased (line 6 in main). The rest of function \text{main} in \(D(X)\) is the same as in \(C^+(X)\): all the relevant monitor states are updated according to \(e\). Function \text{defineNew} in \(D(X)\) first searches for a defined sub-instance of \(\theta\). If such instance exists, \(\theta\) should be defined using it; otherwise, \(\Delta(\theta)\) is set to the initial state. Then \(T(\theta)\) is set to the current timestamp, and the timestamp is incremented. Function \text{defineTo} in \(D(X)\) checks disable and \(T\) as discussed above to decide whether \(\Delta(\theta)\) can be defined using \(\Delta(\theta')\). If \(\Delta(\theta)\) is defined using \(\Delta(\theta')\), \(T(\theta)\) is set to \(T(\theta')\). Both functions then add \(\theta\) to the sets in table \(U\) for the bindings less informative than \(\theta\), as in \(C^+(X)\).

In all of our tested cases \(D(X)\) performs better than \(C^+(X)\); in most cases that \(C^+(X)\) or \(D(X)\) caused notable monitoring overhead, the efficiency of \(D(X)\) is significantly better. For example, in two extreme cases, \(C^+(X)\) could not finish, while

![Algorithm D(X)](image-url)

**Fig. 11.** Optimized Monitoring Algorithm \(D(X)\).
had no problems. Proofs of correctness for $\mathbb{D}(X)$ can be found in [10].

VII. IMPLEMENTATION AND EVALUATION

We implemented code generation for Algorithms $C^+(X)$ and $\mathbb{D}(X)$ in JavaMOP. The indexing technique proposed in [9] is used to implement all the mappings in the algorithms. We evaluated $C^+(X)$, $\mathbb{D}(X)$, and Tracematches on the DaCapo benchmark suite [14]. We omitted other runtime systems because they have been evaluated and compared with either Tracematches or the original JavaMOP algorithm in other papers [5], [6], [9]. Note that Soot [15], 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 [16].

Experimental Settings. Our experiments were performed on a machine with 2GB RAM and a Pentium 4 2.66GHz processor using Ubuntu Linux 7.10. We used version 2006-10 of the DaCapo benchmark suite [14]. The default input for DaCapo was used, and we use the $\text{-converge}$ 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 Fig. 12 should be interpreted as “±3%”. Additional code introduced by the AspectJ weaving process changes the program structure in DaCapo, sometimes causing the benchmark to run a little faster due to better instruction cache layout.

Properties. The following properties used in our experiments are discussed next. They were borrowed from [12], [17], [13]. More examples and results can be found at [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 synchronously;
- 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 [9], [13].

Results and Discussions. Figures 12 and 13 summarize the results of our experiments. Fig. 12 shows the percent overheads of $C^+(X)$, $\mathbb{D}(X)$ (both implemented in JavaMOP), and Tracematches. All the properties were heavily monitored in the experiments. 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 low runtime overhead in most experiments, showing their efficiency. For $\mathbb{D}(X)$, only 7 out of 66 cases caused more than 10% runtime overhead. The numbers for $C^+(X)$ and Tracematches are 9 out of 66 and 15 out of 44, respectively. Fig. 13 shows the comparison among three systems using 7 cases where significant numbers of monitors were created in monitoring. Fig. 13 (A) compares runtime overhead. In all cases, $\mathbb{D}(X)$ outperformed the other two and $C^+(X)$ is better than Tracematches. This shows that JavaMOP provides an efficient solution to monitor parametric specifications despite its generality in terms of specification formalisms. The results also illustrate the effectiveness of the enable set based optimization: on average, the overhead of $\mathbb{D}(X)$ is about 20% less than $C^+(X)$. Moreover, when the property to monitor becomes more complicated, the improvement achieved by the optimization is more significant. In the two extreme cases, namely, blob-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.

Fig. 13 (B) shows the maximum memory usages of our experiments in $\log_{10}$ scale, in Megabytes. It shows that the enable set optimization does not always reduce peak memory usage. In 4 out of 7 cases, the $\mathbb{D}(X)$ has lower peak memory usage than $C^+(X)$. As mentioned, blob-Fig. 13 and pmd-Fig. 13, where $C^+(X)$ did not finish execution, exiting with an out of memory error, $\mathbb{D}(X)$ managed to complete. In the other 3 cases, $C^+(X)$ has slightly lower peak memory usage than $\mathbb{D}(X)$. This is because $C^+(X)$ caused more garbage collections than $\mathbb{D}(X)$. For example, in pmd-SafeIterator, the $C^+(X)$ had 1361 young generation garbage collections while $\mathbb{D}(X)$ had 1167 collections. However, fewer garbage collection cycles contribute to the performance increase of the enable set optimization. This observation also applies to Tracematches: 3 out of 5 cases, Tracematches causes less peak memory usage with more garbage collection but more runtime overhead.

Fig. 13 (C) shows the number of created monitor instances, another important measurement for our approach, also in $\log_{10}$ scale. The number of instance monitors in blob-SafeMapIterator and pmd-SafeMapIterator for $C^+(X)$ 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

6The original JavaMOP is conservatively extended by the new $\mathbb{D}(X)$, in that for properties supported by the original JavaMOP algorithm, $\mathbb{D}(X)$ generates the same monitoring code and instrumentation.
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: 137,880,368 and 165,269,166 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.

VIII. 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