Garbage Collection for Monitoring Parametric Properties*

Dongyun Jin  Patrick O’Neil Meredith  Dennis Griffith  Grigore Roșu
Department of Computer Science, University of Illinois at Urbana-Champaign
{djin3, pmeredit, dgriffi3, grosu}@cs.illinois.edu

Abstract

Parametric properties are behavioral properties over program events that depend on one or more parameters. Parameters are bound to concrete data or objects at runtime, which makes parametric properties particularly suitable for stating multi-object relationships or protocols. Monitoring parametric properties independently of the employed formalism involves slicing traces with respect to parameter instances and sending these slices to appropriate non-parametric monitor instances. The number of such instances is theoretically unbounded and tends to be enormous in practice, to an extent that how to efficiently manage monitor instances has become one of the most challenging problems in runtime verification. The previous formalism-independent approach was only able to do the obvious, namely to garbage collect monitor instances when all bound parameter objects were garbage collected. This led to pathological behaviors where unnecessary monitor instances were kept for the entire length of a program. This paper proposes a new approach to garbage collecting monitor instances. Unnecessary monitor instances are collected lazily to avoid creating undue overhead. This lazy collection, along with some careful engineering, has resulted in RV, the most efficient parametric monitoring system to date. Our evaluation shows that the average overhead of RV in the DaCapo benchmark is 15%, which is two times lower than that of JavaMOP and orders of magnitude lower than that of Tracematches.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Assertion checkers, Class invariants, Formal methods, Reliability; D.2.5 [Software Engineering]: Debugging aids, Error handling and recovery, Monitors; D.3.2 [Programming Languages]: Object-oriented languages; D.3.4 [Programming Languages]: Code generation, Memory management

General Terms Languages, Performance, Reliability, Verification

Keywords runtime verification, runtime monitoring, testing, debugging, aspect-oriented programming, garbage collection

1. Introduction

Monitoring is an effective technique for ensuring software reliability. The well known concept of typestate [31] property can be enforced by using monitoring techniques. Typestates refine the notion of type by stating not only what operations are allowed by a particular object, but also what operations are allowed in what contexts.

Figure 1 shows the typestate description for HASNEXT. The HASNEXT typestate says that it is invalid to call the next() method on an Iterator object when there are no more elements in the underlying Collection, i.e., when hasnext() returns false, or when it is unknown if there are more elements in the Collection, i.e., hasnext() is not called. From the unknown state, it is always an error to call the next() method because such an operation could be unsafe. If hasnext() is called and returns true, it is safe to call next(), so the typestate enters the more state. If, however, the hasnext() method returns false, there are no more elements, and the typestate enters the none state. In the more and none states, calling the hasnext() method provides no new information. It is safe to call next() from the more state, but it becomes unknown if more elements exist, so the typestate reenters the initial unknown state. Finally, calling next() from the none state results in an error.

**Typestate Property Example** - It is straightforward to encode this, and all typestate properties, as particular (one-parameter) parametric properties. Figure 2 shows this property using the RV system presented in this paper. For demonstration purposes, we specify the same property using two different formalisms. The first formalism is a direct translation of the typestate diagram using the finite state machine (FSM) capabilities of RV, and is denoted by the fsm keyword. Each state in the typestate is written as a name followed by its type by stating not only what operations are allowed by a particular object, but also what operations are allowed in what contexts.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI’11, June 4–8, 2011, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0663-8/11/06…$10.00
We wish to catch this behavior because Java uses the extended regular expression (ERE) capabilities of the variables bound by

```
create
update
system, as specified by the Collection
is modified, and then the Iterator is used again. Here we use the extended regular expression (ERE) capabilities of the RV system, as specified by the ere keyword. The occurrence of update before the first create event, in which the Iterator is first created. We wish to catch this behavior because Java Collections do not allow concurrent modification. The JVM usually throws a runtime exception when this occurs, but the exception is not guaranteed to be thrown in a multi-threaded environment.
```

Figure 2. HASNEXT property in RV using FSM and LTL

LTL formula here states that any call to next() must always be immediately preceded (\(*\)) by a call to hasNext() that returned true. The RV system supports multiple logical formalisms, and as this example demonstrates, some formalisms can lead to much more succinct and easy to understand specifications.

The monitored events are prefixed by the keyword event and are encoded using slightly extended AspectJ [26] pointcuts. One such extension is the condition pointcut, which is similar to the if pointcut: it ensures that the given pointcut is only applied if its condition is true, but unlike the if pointcut, it is able to refer to variables bound by returning advice. Thus, the hasNexttrue event is only generated if hasNext() returns true, and the hasNextfalse event is only generated if hasNext() returns false. The per Iterator nature of the HASNEXT typestate is encoded by using one Iterator parameter.

For each property, the code block following it is referred to as a handler. The handler is executed when a condition of its correspond- ing specification is met. For instance, the FSM handler in Figure 2 is executed when the machine enters the error state, while the LTL handler is executed when the LTL formula is violated. Handlers may contain any arbitrary Java code, but here they simply print messages. Such behavior is useful for debugging and testing purposes.

Parametric properties properly generalize typestates, as we shall see, by allowing more parameters. This allows us to specify not only properties about a given object such as the HASNEXT example, but also properties that specify relationships between objects.

General Parametric Property Examples- Figure 3 shows a property for the unsafe use of Collection and Iterator. The property flags it as an error if an Iterator is created, its underlying Collection is modified, and then the Iterator is used again. Here we use the extended regular expression (ERE) capabilities of the RV system, as specified by the ere keyword. The occurrence of update at the beginning of the pattern allows any number of updates before the first create event, in which the Iterator is first created. We wish to catch this behavior because Java Collections do not allow concurrent modification. The JVM usually throws a runtime exception when this occurs, but the exception is not guaranteed to be thrown in a multi-threaded environment.

Figure 3. UNSAFEITER property in RV using the ERE plugin

```
SafeLock(\{\} L, T)
{
    event acquire before(\{\} L, T):
        \{\}
    event release before(\{\} L, T):
        \{\}
    event end after(T):
        \{\}
   redential S
    {\}
}
```

Figure 4 shows a specification for the safe use of reentrant locks, called SAFELOCK, stating that the number of calls of an acquire() in a given method is balanced with the number of calls to release(). This property is parametric both in the Lock in question and in the Thread, and is specified using the context-free grammar (CFG) plugin of the RV system. The events begin and end refer to the beginning and end of every method. The thread pointcut is also an RV extension of standard AspectJ pointcuts that allows for binding the current Thread of execution in the monitored program. The pattern for the specification, prefixed by the keyword cfg, has S for its start symbol. The first symbol seen is always assumed the start symbol. The CFG pattern requires that begin and end events are matched and properly nested with acquire and release events, which must also be matched.

Monitoring parametric properties in their full generality is a complex task. Several parametric monitoring systems such as Eagle [20], J-Lo [11, 12, 30], Tracematches [4, 8], JavaMOP [16, 17, PQL [23], PQL [27], QVM [5], SpoX [24], PoET [22], and RuleR [9] have been proposed in recent years. In parametric monitoring systems, the parameters are dynamically bound to objects at runtime, thus resulting in a potentially unlimited number of monitor instances, one per combination of parameter bindings. The main challenge underlying the monitoring of parametric properties is therefore how to effectively manage these monitor instances, in particular how to efficiently retrieve all the monitor instances interested in an event when it takes place, and how to efficiently garbage collect monitor instances which have become unnecessary.

Earlier attempts such as Tracematches [4, 8] are careful to manage their memory, but hardwire their property specification formalism (regular expression only). JavaMOP [16, 17] is generic in specification formalisms, but, however, it has memory leaks. Due to JavaMOP’s creation of separate monitor instances in order to handle each separate parameters instantiation, recognizing and removing unnecessary monitor instances is quite challenging. JavaMOP is only able to collect a monitor instance when all the bound parameters are garbage collected, which ensures that no event can happen to the corresponding monitor instance. The problem with this method of garbage collection can be clearly seen in the UNSAFEITER property from Figure 3. Because it is the next event at the end of
of the pattern that actually causes the error, there is no way to ever match the pattern if the \texttt{Iterator} bound to a given monitor instance is garbage collected. However, JavaMOP is only able to collect the associated monitor instance if both the Collection and the \texttt{Iterator} are garbage collected. Unfortunately, in most realistic programs, Collections have much longer lifetimes than the \texttt{Iterators} created from them. Because of this, JavaMOP would have large numbers of monitor instances—when monitoring most programs—that could never possibly match the pattern because their bound \texttt{Iterators} had been collected. The RV system, which is presented in this paper, is a commercial grade system developed by Runtime Verification, Inc. RV is able to collect these monitor instances, as well as many others that JavaMOP does not collect.

To collect monitor instances that JavaMOP is unable to collect, we implement, in RV, a means to prune unnecessary monitor instances based on a static analysis of the monitored property. The results of the static analysis, which we refer to as coenables, consist of all those subsets of events which can potentially make update a relevant event for a monitor for UNSAFE\textsc{er}, that is, \{next\}, \{next, update\}, and \{next, create, update\}. Indeed, in any matching trace containing update, the event update is followed by precisely all the events in one of these subsets. Consider now a monitor \( M \) for UNSAFE\textsc{er} corresponding to a particular parameters instance, say \( c \mapsto c_1 \) and \( i \mapsto i_1 \), and suppose that an event update is just being dispatched to \( M \). At this moment, \( M \) knows that it has a future only if all the events in at least one of the coenable sets of update are possible. In particular, if the \texttt{Iterator} \( i_1 \) has already been garbage collected, then \( M \) will never match, since each of the coenable sets of update contains a next, which can only be generated by \( i_1 \). Thus, \( M \) can safely terminate itself and be garbage collected in this situation. Removing unnecessary monitors is still an expensive task, and in the interest of making it as efficient as possible, a lazy collection method is used. This technique makes RV the most efficient parametric monitoring system to date, by a large margin (see Section 5).

Our monitor garbage collection technique is orthogonal to other optimization techniques for parametric monitoring. More precisely, our technique is aimed at improving the base performance of parametric monitoring by means of keeping the number of monitor instances low without relying on (expensive) knowledge about the source program or on minimizing the distance between events and their monitors. Other optimizations can be applied on top of our garbage collection technique and thus start from this base performance and improve it. For example, staged indexing (or decentralized indexing), which has been proposed and implemented in [6, 8, 17], piggy-backs indexing trees onto parameter instances. This reduces the cost of lookup due to better cache locality and fewer hash lookups. Also, significant runtime overhead reductions have been achieved using program static analysis [14, 15, 21, 27], by removing unnecessary instrumentation. RV supports both staged indexing and program static analysis via the Clara approach [15]. Nevertheless, we deliberately disabled these orthogonal optimizations in our evaluation, to properly measure the effectiveness of the proposed garbage collection technique. Enabling these orthogonal optimizations would only hide the inefficiency of base monitoring.

We evaluate our RV garbage collection technique and compare it to those in JavaMOP and Tracematches in Section 5. We picked these two systems for comparison because they are known for their efficiency (the best so far). The average overhead of RV in version 9.12 of the DaCapo [10] benchmark suite is 15%, even with no static or decentralized indexing optimizations, which is two times lower than 33% of JavaMOP and nine times lower than the 142% of Tracematches disregarding those cases where Tracematches failed to terminate. Even the largest overhead of \( M \) in two versions of DaCapo, from UNSAFE\textsc{er}-\textsc{blot}, is only 251%, while in JavaMOP, 7 cases show overhead higher than 251%, and in Tracematches, 20 cases show higher overhead and 9 cases do not terminate.

**Outline**

The rest of this paper is as follows: Section 2 provides a brief overview of parametric monitoring; Section 3 explains the theory of the coenables used for pruning unnecessary monitor instances, and shows some examples; Section 4 discusses our data structures to efficiently garbage collect monitors by using coenables sets, as well as how the coenable sets are actually used during the monitoring process; Section 5 presents our experimental data; and Section 6 provides some concluding remarks.

### 2. Parametric Properties and Monitoring

To explain the garbage collection of unnecessary monitor instances, we first introduce some background theory on parametric monitoring. For consistency, we follow the notation and terminology recently proposed by the JavaMOP authors in [18]. We begin by introducing the notions of event, trace, and property, first non-parametric and then parametric. Trace slicing is then defined as a reduction operation that forgets events that are unrelated to the given parameter instance.

**Definition 1.** Let \( E \) be a finite set of (non-parametric) events, called base events or simply events. An \( E \)-trace, or simply a (non-parametric) trace when \( E \) is understood or not important, 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 \( e \in w \). \( e \) is the empty trace.

For UNSAFE\textsc{er} in Section 1, the set of events \( E \) is \{create, update, next\}, and a possible trace is “create next update next”.

**Definition 2.** An \( E \)-property \( P \), or simply a (base or non-parametric) property, is a function \( P: E^* \rightarrow \mathcal{C} \) partitioning the set of traces into (verdict) categories \( \mathcal{C} \). In general, \( \mathcal{C} \) may be any set.

Consider again UNSAFE\textsc{er}. The match traces are those matching the pattern, e.g., “create next update next”. There are also traces that have not matched yet, but may still match in the future, such as “update create”, which we call \( ? \) (unknown) traces. Lastly, there are traces that may never match again, such as “create update next”, which we refer to as fail traces. Thus we pick \( \mathcal{C} \) to be the set \{match, fail, \}?\}, and define its property \( P_{\text{UNSAFE\textsc{er}}}: E^* \rightarrow \mathcal{C} \) as follows: \( P_{\text{UNSAFE\textsc{er}}}(w) = \text{match} \) if \( w \) is in the language of the UNSAFE\textsc{er} ere, \( P_{\text{UNSAFE\textsc{er}}}(w) = \? \) if \( w \) is a prefix of a string in the language of the ere, and \( P_{\text{UNSAFE\textsc{er}}}(w) = \text{fail} \) otherwise.

We next extend the above definitions to the parametric case. Let \([A \rightarrow B]\) be the set of total functions, and let \([A \rightarrow B]\) be the set of partial functions from \( A \) to \( B \).

**Definition 3.** (Parametric events and traces). Let \( X \) be a finite set of parameters and let \( V \) be a set of corresponding parameter values. If \( E \) is a set of base events like in Definition 1, then let \( E(X) \) be the set of corresponding parametric events \( e(\theta) \), where \( e \) is a base event in \( E \) and \( \theta \) is a partial function in \([X \rightarrow V]\) Partial functions \( \theta \) in \([X \rightarrow V]\) are called parameter instances. A parametric trace is a trace with events in \( E(X) \), that is, a word in \( E(X)^* \).

A parametric trace for UNSAFE\textsc{er} could be “update(\( c \mapsto c_1 \)) update(\( c \mapsto c_2 \)) create(\( c \mapsto c_1, i \mapsto i_1 \)) next(\( i \mapsto i_1 \))”. To simplify writing we often assume the parameter set implicit, as in the following, which is the same trace: “update(\( c_1 \)) update(\( c_2 \)) create(\( c_1, i_1 \)) next(\( i_1 \))”.

**Definition 4.** Let \( X \) be a finite set of parameters. If \( E \) is a set of base events like in Definition 1, we define a parametric event definition, or event definition for short, as a function \( D: E \rightarrow \mathcal{P}(X) \), where
\( P \) is the power set, that maps each event \( e \) to a set of parameters \( D(e) \) that will be instantiated by \( e \) at runtime. \( D \) is extended to \( E^* \) as \( D(e) = \emptyset \) and \( D(ew) = D(e) \cup D(w) \), and to \( P(E) \) as \( D(\emptyset) = \emptyset \) and \( D(E) \cup D(\emptyset) = D(E) \cup D(\emptyset) \). Parametric event \( e(\theta) \) is \( D \)-consistent if \( \text{dom}(\emptyset) = D(e) \), Parametric trace \( \tau \) is \( D \)-consistent if \( e(\theta) \) is \( D \)-consistent for each \( e(\theta) \) in \( \tau \).

The UnsafeIfTer property contains the parametric event definition \( D(\text{create}) = \{c, i\} \), \( D(\text{update}) = \{c\} \), \( D(\text{next}) = \{\} \). It states that, for example, parameters \( c \) and \( i \) will be instantiated at runtime when a parametric event \( e(\theta) \) is received. For a trace "create update", \( D(\text{create update}) = \{c, i\} \).

**Definition 5.** \( \theta, \theta' \in [A \rightarrow B] \) are compatible if for any \( x \in \text{dom}(\theta) \cap \text{dom}(\theta') \), \( \theta(x) = \theta'(x) \). We can combine compatible instances \( \theta \) and \( \theta' \), written \( \theta \sqcap \theta' \), as follows:

\[
(\theta \sqcap \theta')(x) =
\begin{cases}
    \theta(x), & \text{if } \theta(x) \text{ is defined} \\
    \theta'(x), & \text{if } \theta'(x) \text{ is defined} \\
    \text{undefined otherwise}
\end{cases}
\]

\( \theta \sqcap \theta' \) also is called the least upper bound (lub) of \( \theta \) and \( \theta' \). \( \theta \) is less informative than \( \theta' \), written \( \theta' \sqsupseteq \theta \), if for any \( x \in \text{dom}(\theta) \), if \( \theta(x) \) is defined then \( \theta'(x) \) is also defined and \( \theta(x) = \theta'(x) \). \( \sqcap \) is extended to \( P(\{X \mapsto V\}) \) in the natural way. Here \( P(\emptyset) \) is the finite power set.

**Definition 6.** *(Trace slicing)* Given parametric trace \( \tau \in E(X)^* \) and \( \emptyset \in [X \mapsto V] \), let the \( \theta \)-trace slice \( \tau|_{\theta} \in E^* \) be the non-parametric trace defined as:

- \( \epsilon|_{\emptyset} = \epsilon \) (recall that \( \epsilon \) is the empty trace)
- \( (\tau e(\theta'))|_{\emptyset} =\) if \( \emptyset \sqsubseteq \theta' \) \( (\tau|_{\emptyset}) e \) or \( (\tau e(\emptyset))|_{\emptyset} \) otherwise

The trace slice \( \tau|_{\emptyset} \) first filters out all the parametric events that are not relevant for the instance \( \emptyset \), i.e., which contain instances of parameters that \( \emptyset \) does not care about, and then, for the remaining events relevant to \( \emptyset \), it forgets the parameters so that the trace can be checked against base, non-parametric properties. It is crucial to discard events from parameter instances that are not relevant to \( \emptyset \) during the slicing, including those more informative than \( \emptyset \). Referring back to our parametric trace from above, the non-parametric trace slice for parameter instance \( \{c_2\} \) is "updated", that for \( \{c_1\} \) is "update", the slice for \( \{c_1, i_1\} \) is "update next", and the slice for \( \{i_1\} \) is "next".

**Definition 7.** Let \( X \) be a finite set of parameters together with their corresponding parameter values \( V \). 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 verdict categories \( [[X \mapsto V] \rightarrow C] \))

\[
\Lambda X.P : E(X)^* \rightarrow [[X \mapsto V] \rightarrow C]
\]

where \( \Lambda X.P(\tau)(\emptyset) = P(\tau|_{\emptyset}) \) for each \( \tau \in E(X)^*, \emptyset \in [X \mapsto V] \).

A parametric property is therefore similar to a normal property, but one partitioning parametric traces in \( E(X)^* \) into verdict categories in \( [[X \mapsto V] \rightarrow C] \), that is, original (as in the non-parametric property) verdict categories indexed by parameter instances. This allows the parametric property to associate an original category for each parameter instance from \( [X \mapsto V] \).

Next we define monitors and parametric monitors. Like for parametric properties, which are just properties over parametric traces, parametric monitors are also just monitors, but for parametric events and with instance-indexed states and verdict categories.

**Definition 8.** A monitor \( M \) is a tuple \( (S, E, C, i, \sigma, \gamma) \), where \( S \) is the set of states, \( E \) is the set of input events, \( C \) is the set of verdict categories, \( i \in S \) is the initial state, \( \sigma : S \times E \rightarrow S \) is the transition function, and \( \gamma : S \rightarrow C \) is the verdict function. The transition function is extended to handle traces, i.e., \( \sigma : S \times E^* \rightarrow S \) where

\[
\sigma(s,e) = s \text{ and } \sigma(s,ew) = \sigma(s,\epsilon) \cdot \sigma(s,e) \text{ for each } s \in S, e \in E, \epsilon \in E^*.
\]

**Algorithm MONITOR**

\[
\text{function main}(\tau) \begin{array}{l}
1 \quad \Delta \leftarrow \bot, \Delta(\emptyset) \leftarrow \emptyset, \emptyset \leftarrow \{\bot\} \\
2 \quad \text{for each } e(\theta) \text{ in order in } \tau \text{ do} \\
3 \quad \text{if } e(\theta) \text{ then } \\
4 \quad \text{for each } \emptyset' \in \{\emptyset\} \sqcup \theta \text{ do} \\
5 \quad \Delta(\emptyset') \leftarrow \gamma(\Delta(\emptyset'), e) \\
6 \quad \text{endfor} \\
7 \text{endfor} \\
8 \end{array}
\]

**Figure 5.** Monitoring Algorithm

\[
\sigma(s, e) = s \text{ and } \sigma(s, ew) = \sigma(s, \epsilon) \cdot \sigma(s, e) \text{ for each } s \in S, e \in E, \epsilon \in E^*.
\]

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 9.** Given parameters \( X \) with corresponding values \( V \) and monitor \( M = (S, E, C, i, \sigma, \gamma) \) is a monitor for property \( P : E^* \rightarrow C \) if \( \gamma(\sigma(s, e)) = P(s) \) for each \( s \in S \). Monitor \( M \) defines the property \( P_M : E^* \rightarrow C \) with \( P_M(w) = \gamma(\sigma(i, w)) \). Monitors \( M \) and \( M' \) are equivalent iff \( P_M = P_{M'} \).

Therefore, a 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 category of \( M \) per instance). Intuitively, one can think of a parametric monitor as a collection of "monitor instances". Each monitor instance, which is indexed by a parameter instance, keeps track of the state of the corresponding instance from \( [X \mapsto V] \).

The rule for \( \Lambda X.\gamma \) can be read as stating that when an event with parameter instance \( \emptyset' \) is evaluated, it updates the state for all monitor instances more informative than the instance for \( \emptyset' \), and the instance for \( \emptyset' \) itself, leaving all other monitor instances untouched. The rule for \( \Lambda X.\gamma \) simply states that \( \gamma \) is applied to a state, as normal, but the state is found by looking up the state of the monitor instance for \( \emptyset \). One of the major results in [18] states that if \( M \) is a monitor for \( P \), parametric monitor \( \Lambda X.M \) is a monitor for the parametric property \( \Lambda X.P \).

Figure 5 shows the basic abstract monitoring algorithm for parametric properties from [18]. Given parametric property \( \Lambda X.P \) and \( M \) a monitor for \( P \), \( \text{MONITOR}(M) \) yields a monitor that is equivalent to \( \Lambda X.M \), that is, a monitor for \( \Lambda X.P \). The functions \( [[X \mapsto V] \rightarrow S] \) and \( [[X \mapsto V] \rightarrow C] \) of \( \Lambda X.M \) are encoded by \( \text{MONITOR}(M) \) as tables \( \Delta \) and \( \Gamma \) with entries indexed by parameter instances in \( [X \mapsto V] \) and with contents states in \( S \) and verdict categories in \( C \), respectively. Such tables will have finite entries because each event \( e \) binds only a finite number of parameters defined by \( D(e) \).

The monitoring algorithm first clears \( \Delta \), which contains the monitor state for each parameter instance, then assigns \( i \), the initial state, to \( \Delta(\bot) \). \( \emptyset \), which contains all known parameter instances, is initialized to contain only the empty partial function \( \bot \). For
When monitoring parametric properties, it is easy to generate a large number of monitor instances. Specifically, for each parameter instance \( \theta \) with all the previously known compatible parameter instances (line 3), it updates the state of every one of these compatible parameter instances \( \theta' \) with the state, transitioned by event \( e \), of the “monitor instance” corresponding to the “largest” parameter instance less than or equal to \( \theta' \) (line 4). At the same time, we also calculate the verdict category corresponding to that monitor instance and store it in a table \( \Gamma \) (line 5). Rather than storing a whole slice as in Definition 6, the knowledge of the slice is encoded in the state of the monitor instance for \( \theta' \). After the algorithm completes, \( \Gamma \) contains the verdict category for each possible trace slice. An actual implementation is free to report a verdict category of interest (e.g., match or fail) as soon as it is discovered.

3. Coenable Sets

When monitoring parametric properties, it is easy to generate a large number of monitor instances. For example, as seen in Section 5, the program bloat generates 1.9 million monitor instances when monitored for the UNSAFEITER property. After some time, some of these monitor instances may become unnecessary, e.g., because they have no hope of reaching a verdict category in \( G \). Indeed, as seen in Section 5, the code executed becomes an 8.1 million slice of code. For some monitor instances, this set of monitors may become unnecessary. Then, et al. [19] proposed a formalism-independent method, called “coenable sets”, to avoid unnecessarily creating monitors that will never trigger. Here we show how a dual method can be derived to avoid unnecessarily retaining monitors that will never trigger. Computing the coenable sets is expected to be a quick and efficient operation in practice, because they are a function of the specification to monitor (which is expected to be small) and not of the program (which is expected to be large).

Definition 10. Given \( w \in E^* \) and \( e, e' \in w \), we let \( e \leadsto w e' \) denote that \( e' \) occurs after \( e \) in \( w \). Let \( \text{COENABLE}_{w}(e) = \{ e' \mid e \leadsto w e' \} \) be the trace coenable set of \( e \). Given property \( P : E^* \to C \) and a subset of verdict categories of interest (or goal) \( G \subseteq C \), the property coenable set is defined as the map \( \text{COENABLE}_{P,G} : E \to P(P(E)) \) where \( \text{COENABLE}_{P,G}(e) = \{ \text{COENABLE}_{w}(e) \mid w \in E^* \text{ s.t. } P(w) \in G, e \in w, \text{COENABLE}_{w}(e) \neq \emptyset \} \) for each \( e \in E \). Intuitively, if event \( e \) is encountered during monitoring, but none of the event sets of \( \text{COENABLE}_{P,G}(e) \) are possible in the future, it is impossible to reach any verdict category in \( G \), so a monitor for \( P \) observing \( e \) will never trigger. We drop all \( \emptyset \)'s from \( \text{COENABLE}_{P,G}(e) \) because they can cause monitor instances to be retained that are unnecessary. An \( \emptyset \) in \( \text{COENABLE}_{P,G}(e) \) means that the trace suffix consisting of only the event \( e \) can lead to a category in \( G \) for some trace prefix. However, our interest is in the ability to reach \( G \) again in the future. If there is a trace suffix that can lead to a state in \( G \) from \( e \), then its events will be added to \( \text{COENABLE}_{P,G}(e) \). If there is no trace suffix that can lead back to a state in \( G \), there is no reason to maintain the monitor instance after it has executed the proper handler due to the occurrence of \( e \).

FSM Example We define finite state machines in the spirit of Definition 8. A finite state machine is a tuple \((S, E, C, I, \sigma, \gamma)\) where \( E \) is a finite alphabet, \( S \) is a finite set of states, \( I \in S \) is the initial state, \( \sigma : S \times E \to S \) is a partial transition function, \( C \) a set of verdict categories, and \( \gamma : S \to C \) the verdict function. The property monitored by an FSM classifies a trace \( w \) into \( \gamma(\sigma(1, w)) \), where \( \sigma \) is extended to strings in the natural way, and fail if \( \sigma(1, w) \) is undefined.

We can find \( \text{COENABLE}_{P,G} \) for the property monitored by an FSM, by the least fixed point of the following equations. Recall that \( G \subseteq C \) is the set of verdict categories of interest:

\[
\text{SEEABLE}(s) = \bigcup_{s(e,s') = s} \{ s \} \cup \{ T \mid T \in \text{SEEABLE}(s') \}
\]

\[
\text{COENABLE}_{P,G}(e) = \bigcup_{s(e,s') = s} \text{SEEABLE}(s')
\]

We can use the equations above to generate coenable sets for our example from Figure 3. One need only generate a finite state machine from the property’s ERE. For \( P = \text{UNSAFEITER} \) and \( G = \{ \text{match} \} \), the \( \text{COENABLE}_{P,G} \) sets are:

\[
\text{COENABLE}_{P,G}(\text{create}) = \{ \{ \text{next, update} \} \}
\]

\[
\text{COENABLE}_{P,G}(\text{update}) = \{ \{ \text{next}, \{ \text{next, update} \} \} \}
\]

\[
\text{COENABLE}_{P,G}(\text{next}) = \{ \{ \text{next, update} \} \}
\]

Note that if we did not remove \( \emptyset \)'s, \( \text{COENABLE}_{P,G}(\text{next}) \) would contain \( \emptyset \). Each inner set can be thought of as a conjunction of events that must occur at least once for a verdict category in \( G \) to be reachable, while the outer sets are a disjunction (see Section 4.2.2). For example, if the event seen by monitor instance \( M \) is update and next can still be seen at some future point, then \( M \) is still necessary. Likewise, if the event seen by \( M \) is next, then both next and update must be possible for \( M \) to ever match. In particular, if the corresponding Collection object instance is already dead then we know that the event update will never be possible, so we can safely garbage collect \( M \). Definition 11 formalizes this notion.

CFG Example A CFG is a tuple \((N, E, S, I, II)\) where \( N \) is a finite set of nonterminals, \( E \) is a finite set of terminals, \( S \in N \) is the initial nonterminal, and \( II \) is a set of productions of the form \( A \to B \) where \( A \in N \) and \( B \in (N \cup E)^* \). The CFG classifies traces that are in the language of the grammar into the verdict category match.

For a CFG, to compute \( \text{COENABLE}_{P,\{\text{match}\}} \), we find the least fixed point of the following equations:

\[
\text{G}(\emptyset) = \{ \emptyset \}
\]

\[
\text{G}(e) = \{ \emptyset \} \cup \bigcup_{s(e,s') = s} \{ s \} \cup \{ T \mid T \in \text{G}(s') \}
\]

\[
\text{G}(A) = \bigcup_{A \to B} \text{G}(B)
\]

\[
\text{C}(e) = \bigcup_{T_1 \cup T_2 \mid T_1 \in \text{G}(A)} \{ T_1 \to T_2 \mid A \to B \}
\]

\[
\text{COENABLE}_{P,\{\text{match}\}}(e) = \text{C}(e)
\]

Informally, \( \text{G}(A) \) is the set of events generated by the CFG, if the symbol \( A \) were used as the initial nonterminal of the CFG. The equation \( \text{G}(A) = \bigcup_{B \in \text{G}(B)} \text{G}(B) \) generalizes this notion to entire traces of symbols (where symbols are either events or non-terminals). \( \text{C} \) is the coenable sets function generalized to traces that include both non-terminals and events. For a production, \( A \to B \), \( \text{C}(B) \) needs to cope with the fact that \( A \) has its own coenable sets. Thus its definition unifies possible coenable sets of \( A \) with the sets of symbols that are generated by \( B \). The rest of the \( \text{CFG} \) only needs to know coenable sets for events so coenables is just the restriction of \( C \) to events.

Definition 11. Given property \( P : E^* \to C \), goal \( G \subseteq C \), set of verdict categories \( X \) and event definition \( D : E \to P(X) \) (see Definition 4), the property parameter coenable set is defined as the map \( \text{COENABLE}_{P,G}^X : E \to P(P(X)) \) where \( \text{COENABLE}_{P,G}^X(e) = \{ D(E) \mid E \in \text{COENABLE}_{P,G}(e) \} \) for each \( e \in E \).

The \( \text{COENABLE}_{P,G}^X \) sets tell us which parameter objects must be alive for a verdict category in \( G \) to be reachable. For \( P = \text{UNSAFEITER} \) and \( G = \{ \text{match} \} \), and \( X = \{ c, i \} \), the \( \text{COENABLE}_{P,G}^X \)
sets are:

\[
\text{COENABLE}_{X,G}^\text{create}(c) = \{\{c, i\}\}
\]
\[
\text{COENABLE}_{X,G}^\text{update}(i) = \{\{i, \{c, i\}\}\}
\]
\[
\text{COENABLE}_{X,G}^\text{next}(c) = \{\{c, i\}\}
\]

Now with the COENABLE sets we can explicitly decide when a monitor instance may be collected. For example, in UNSAFEITER we know that if, at any time, the Iterator bound to \(i\) is garbage collected, then a match can never occur because \(i\) occurs in every one of the inner sets. This makes sense because the event that causes a match in the UNSAFEITER pattern is use of the Iterator. As mentioned in Section 1, this situation could produce a very large memory leak in JavaMOP [17] where long living collections would cause monitor instances for dead Iterators to be retained because it could not remove a monitor instance unless all bound parameter objects were collected. We prove this concept by showing that certain parameters specified by COENABLE sets must be able to occur in \(w\) for a verdict category to be reached.

**Theorem 1.** Consider the same assumptions as in Definition 11, and a trace slice \(w \in \mathcal{L}\). If for each \(Y \in \text{COENABLE}_{X,G}(e)\) there exists some \(Y \in \mathcal{D}\) such that \(Y \notin \mathcal{D}(w')\) then \(P(w') \notin G\).

**Proof.** Suppose, for the sake of contradiction, that \(P(w') \in G\) and each \(Y \in \text{COENABLE}_{X,G}(e)\) contains a \(y\) such that \(y \notin \mathcal{D}(w')\). By Definition 10, because \(P(w') \in G\) there must be some \(E \in \text{COENABLE}_{X,G}(e)\) that contains exactly those events in \(w'\). Then, by Definition 11, there must be \(Y \in \text{COENABLE}_{X,G}(e)\) containing exactly the parameters in \(\mathcal{D}(w')\). Contradiction.

**Discussion** The COENABLE sets are a conservative approximation of the situations in which a monitor instance may be collected. From Definition 6 we know that an event \(e\) where \(x \in D(e)\) can only occur in a trace-slice \(x\) if \(\theta(x)\) is still alive in the system. If \(\theta(x)\) has been garbage collected, there is no way for any \(e\) with \(x \in D(e)\) to occur in trace slice for \(\theta\). This is precisely how monitoring arrives in the situation presented in Theorem 1, where all possible suffixes \(w'\) of the trace slice \(w\) do not contain at least one parameter in each set of the COENABLE sets, and it becomes impossible to reach a verdict category in \(G\). Clearly, if it is impossible for the \(\theta\) trace slice to ever reach a verdict category in \(G\), there is no reason to keep the monitor instance for \(\theta\).

The Tracematches system uses a more precise formulation, which is similar, but based on the state of the monitor. Intuitively, the Tracematches garbage collection technique can be thought of as coenable sets indexed by state rather than events, but the formulation as presented in [8] is considerably different. While theirs is more precise, our empirical results, presented in Section 5, show that the coenable set technique is able to reduce memory usage in the RV system to comparable levels with Tracematches, while the RV system has considerably lower runtime overhead. More importantly, the Tracematches garbage collection technique is limited to finite logics, such as the regular expressions of Tracematches. However, our coenable approach is extensible to any underlying monitor implementation. We have a coenable sets generation algorithm for the context-free grammar plugin. A static state-based technique, such as the one used by Tracematches, could not be used for context-free properties because the state space is unbounded.

The coenable technique claims much more memory than JavaMOP’s garbage collection, which, as already explained, has to wait for all bound parameter objects to be collected (see Section 5).

**4. Implementation**

The data structures used by previous runtime monitoring systems [4, 17, 20] are not sufficient for efficient garbage collection of monitor instances. The challenge is how to efficiently garbage collect unnecessary monitor instances that are contained in the data structures. Using the standard data structures of previous systems, the overhead of instance removal easily overwhelms the benefit of having fewer instances. Our specialized data structures, introduced here, track the garbage collection of parameter objects and remove unnecessary monitor instances when discovered using coenable sets (Section 3). In this section, we present the modified indexing trees used by RV as well as the mechanism by which unnecessary monitors are garbage collected.

**4.1 Indexing Trees**

The RV system builds upon the Indexing Tree technique of the JavaMOP system presented in [17]. The indexing trees are an efficient means to represent the tables \(\Delta\) and \(\Gamma\) from the monitoring algorithm in Figure 5. Locating the correct monitor instances to update for each received event is one of the most important and expensive tasks of a runtime monitoring system that supports formalism-independent parametric properties like JavaMOP and RV. Whenever a parametric event \(e(\theta)\) is processed, all monitor instances corresponding to parameter instances more informative than \(\theta\) must be updated. Thus we need a mapping from parameter instances to sets of monitor instances more informative than that instance. Each value in the map is either the next level of the tree or, at the leaves, the appropriate set of monitor instances. Once we have this set we update all the contained monitor instances.

For example, processing update\((c_2)\) for UNSAFEITER (Figure 3) requires that we update the monitor instances that are more informative than \((c_2)\). Thus we lookup \((c_2)\) in the \((c)\)-tree of Figure 6 to find the set of monitor instances more informative than \((c_2)\) and update each instance. Multiple indexing trees can exist since each event may contain a different subset of parameters; each subset of possible parameters receives its own tree. As an example, for UNSAFEITER we have a \((c)\)-tree, an \((i)\)-tree, and a \((c, i)\)-tree.

**4.2 Collecting Unnecessary Monitors**

There are two performance benefits to garbage collecting unnecessary monitors: reduced memory usage, and reduction in the time needed to update monitor instances because many of the monitor instances that would be updated are no longer necessary. As an example of the latter, consider UNSAFEITER again. If we have a monitor instance for \((c_1, i_1)\) and \(i_1\) is garbage collected, even

![Figure 6. Indexing trees for UNSAFEITER](image-url)
 though it is impossible to match the UNSAFEITER pattern, all future update events are sent to \( \langle c_1, i_1 \rangle \). Unfortunately, collecting monitor instances introduces overhead; we must keep this overhead low so that it does not outweigh the benefits of garbage collection.

Eager garbage collection of unnecessary monitors introduces a very large amount of runtime overhead, which almost always overwhelms any benefits. This is because eager collection requires propagating the information regarding liveness of parameter objects to monitor instances far too frequently. Additionally, eager collection can result in removing instances from some data structures that will never be used again.

Therefore, we use a lazy garbage collection scheme. We iterate monitor instances and propagate the information of garbage collections of parameter objects lazily, and we remove unnecessary monitors lazily. When an indexing tree containing a garbage collected parameter object is accessed and the tree detects this, it inform the all relevant monitor instances contained within itself. Then, the monitor instance decides if it can still possibly reach a verdict category in \( G \) in the absence of the parameter object that has been garbage collected. Later when more space is needed in the data structure or when monitor instances are updated, we remove monitor instances from the accessed data structure but not from other data structures. A monitor instance is garbage collected when it is removed from all data structures. This is similar to mark-and-sweep garbage collection. If a data structure itself is garbage collected, any contained monitor instances never need to be garbage collected separately. The next sections explain this process in detail.

4.2.1 Parameter Object Garbage Collection Notification

Propagation of parameter object garbage collection information starts from the mappings in the indexing tree. The mappings used

---

**Figure 7.** (A) Notifying monitors for garbage collected \( \langle c_2 \rangle \) in the \( \langle c \rangle \)-tree. (B) Cleaning up the broken mapping in the \( \langle c \rangle \)-tree

---

in the RV are implemented as a class called RVMap. RVMap uses WeakReferences for its keys. A WeakReference in Java does not stop the garbage collector from collecting its referent; when the referent is garbage collected, the WeakReference points to null. Whenever an operation (put or get) is performed on an RVMap—or the hash table underlying the map needs to be expanded to store more entries—it looks through a subset of its entries for keys with null referents. When there is a key with a null referent due to a garbage collection, RVMap notifies all of the monitor instances below itself in the indexing tree. For example, Figure 7 (A) shows a possible scenario where \( \langle c_2 \rangle \) is garbage collected and the \( \langle c \rangle \)-tree is accessed. The \( \langle c \rangle \)-tree notifies all of the monitor instances below \( \langle c_2 \rangle \).

4.2.2 Determining When Monitor Instances are Unnecessary

When a monitor is notified of a newly garbage collected parameter object, it decides whether it can still reach a verdict category of interest in the absence of garbage collected parameter objects by using the coenable sets introduced in Section 3. Each monitor instance stores the last event it receives, \( e \), so that it may check \( \text{COENABLE}_X^P(e) \), when this notification takes place. The monitor instance need simply check if all the parameter objects of any set in \( \text{COENABLE}_X^P(e) \) are alive. RV statically translates \( \text{COENABLE}_X^P(e) \) to a minimized boolean formula to make this check as efficient as possible:

\[
\text{ALIVENESS}(e) = \bigvee_{S \in \text{COENABLE}_X^P(e)} \left( \bigwedge_{x \in S} \text{live}_x \right)
\]

where \( \text{live}_x \) is a boolean that is true only if the parameter object of parameter \( x \) has not been garbage collected. Then, \( \text{ALIVENESS}(e) \) is true only if the monitor is necessary. Maintaining \( \text{live}_x \) variables in a given monitor instance for each parameter and checking the generated boolean expression at runtime is sufficient for determining when said instance becomes unnecessary.

Continuing our example from the last section, the monitor instances notified of garbage collected parameters in Figure 7 (A) check their \( \text{ALIVENESS} \) to determine if they are unnecessary. Here, \( m_1 \) and \( m_3 \) are unnecessary and therefore marked. Note that the set under \( \langle c_2 \rangle \) is not altered because other RVMaps in the index tree still point to it. In Figure 7 (B), the RVMap removed the broken map entry index by \( c_2 \). \( m_1 \) and \( m_3 \) will be removed at some future time when the \( \langle c, i \rangle \)-tree or \( \langle i \rangle \)-tree are accessed or expanded, as we explain in the next section.

4.2.3 Removing Unnecessary Monitor Instances

Monitor instances are removed lazily because in many cases the maps and sets containing monitor instances flagged for removal may be garbage collected themselves. Eager removal would result in unnecessary work in such cases. For example, in Figure 7 (B), if the \( \langle c_2 \rangle \)-subtree in the \( \langle c \rangle \)-tree is going to be garbage collected, there is no reason to remove flagged monitor instances from it.

Unnecessary monitor instances are only removed when an indexing tree is accessed. Whenever an RVMap looks for keys with null referents it also checks the values of mappings which do not

---

**Figure 8.** A compaction in RVSet when some monitor instances are collectable

---
Figure 9. Comparison of Tracematches (TM), JavaMOP (MOP), and RV: (A) average percent runtime overhead; (B) total peak memory usage in MB. (convergence within 3%, ∞: not terminated after 1 hour)

5. Evaluation of the RV System

We evaluate our formalism-independent garbage collection for parametric monitoring implemented into RV. Also, we compare the performance of RV to JavaMOP and Tracematches, two of the most optimized monitoring systems in runtime and memory, respectively.

5.1 Experimental Settings

For our experiments, we used a Pentium 4 2.66GHz / 2GB RAM / Ubuntu 9.10 machine and version 9.12 of the DaCapo (DaCapo 9.12) benchmark suite [10], the most up-to-date version. We also present the result from the previous version, 2006-10 MR2 of DaCapo (DaCapo 2006-10), but only for the bloat and jython benchmarks. DaCapo 9.12 does not provide the bloat benchmark from the DaCapo 2006-10, which we favor because it generates large overheads when monitoring Iterator-based properties. The bloat benchmark with the UNSAFEITER specification causes 1919% runtime overhead (i.e., 192 times slower) and uses 7.7MB of heap memory in Tracematches, and causes 569% runtime overhead and uses 147MB in JavaMOP, while the original program uses only 4.9MB. Also, although the DaCapo 9.12 provides jython, Tracematches cannot instrument jython due to an error. Thus, we present the result of jython from the DaCapo 2006-10. The default data input for DaCapo was used and the -converge option to obtain the numbers after convergence within ±3%. We also looked into other benchmarks including Java Grande [29] and SPECjvm 2008 [2], and saw little to no overhead even with our Iterator-based properties. Instrumentation introduces a different garbage collection behavior in the monitored program, sometimes causing the program to slightly outperform the original program; this accounts for the negative overheads seen in both runtime and memory.

We used the Sun JVM 1.6.0 for the entire evaluation. The AspectJ compiler (ajc) version 1.6.4 is used for weaving the aspects generated by JavaMOP and RV into the target benchmarks. Another AspectJ compiler, abc [7] 1.3.0, is used for weaving Tracematches properties because Tracematches is part of abc and does not work with ajc. For JavaMOP, we used the most recent release version, 2.1.2, from the JavaMOP website [1]. For Tracematches, we used the most recent release version, 1.3.0, from [3], which is included in the abc compiler as an extension. To figure out the reason that some examples do not terminate when using Tracematches, we also used the abc compiler for weaving aspects generated from RV properties. Note that RV is AspectJ compiler independent. RV shows similar overheads and terminates on all examples when using the abc compiler for weaving as when ajc is used. Because the overheads are similar, we do not present the results of using abc to weave RV generated aspects in this paper. However, using abc to weave RV properties confirms that the high overhead and non-termination come from Tracematches itself, not from the abc compiler.

The following properties are used in our experiments. They were borrowed from [13, 14, 19, 28].

• HASNEXT: Do not use the next element in an Iterator without checking for the existence of it (see Figure 2);
• UNSAFEITER: Do not update a Collection when using the Iterator interface to iterate its elements (see Figure 3);
All of them are tested on.

Therefore, we mainly discuss those three examples in this section, namely, the cases where

JavaMOP shows over 440% runtime overhead and Tracematches always shows over 1350%.

With avora, on average, RV shows 62% runtime overhead, while

JavaMOP shows 139% runtime overhead and Tracematches shows 203% and hangs for UnsafeMAP.

With pmd, on average, RV shows 94% runtime overhead, while JavaMOP shows 231% runtime overhead and Tracematches shows 1139% and hangs for UnsafeMAP and UnsafeSyncMap.

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

Figure 9 (B) shows the peak memory usage of the three systems. RV has lower peak memory usage than JavaMOP in most cases. The cases where RV does not show lower peak memory usage are within the limits of expected memory jitter. However, memory usage of RV is still higher than the memory usage of Tracematches in some cases. Tracematches has several finite automata specific memory optimizations [8], which cannot be implemented in a formalism-independent system like RV. Although Tracematches is sometimes more memory efficient, it shows prohibitive runtime overhead monitoring both pmd and sunflow. There is a trade-off between memory usage and runtime overhead. If RV more actively removes terminated monitors, memory usage will be lower, at the cost of runtime performance. Overall, our monitor termination optimization achieves the most efficient parametric monitoring system with reasonable memory performance.

Figure 10 shows the number of triggered events, of created monitors, and of flagged monitors for each property, while JavaMOP always shows over 440% runtime overhead and Tracematches always shows over 1350%.

Also, the cases where RV does not show lower peak memory usage are within the limits of expected memory jitter. However, memory usage of RV is still higher than the memory usage of Tracematches in some cases. Tracematches has several finite automata specific memory optimizations [8], which cannot be implemented in a formalism-independent system like RV. Although Tracematches is sometimes more memory efficient, it shows prohibitive runtime overhead monitoring both pmd and sunflow. There is a trade-off between memory usage and runtime overhead. If RV more actively removes terminated monitors, memory usage will be lower, at the cost of runtime performance. Overall, our monitor termination optimization achieves the most efficient parametric monitoring system with reasonable memory performance.

Figure 10 (B) shows the peak memory usage of the three systems. RV has lower peak memory usage than JavaMOP in most cases. The cases where RV does not show lower peak memory usage are within the limits of expected memory jitter. However, memory usage of RV is still higher than the memory usage of Tracematches in some cases. Tracematches has several finite automata specific memory optimizations [8], which cannot be implemented in a formalism-independent system like RV. Although Tracematches is sometimes more memory efficient, it shows prohibitive runtime overhead monitoring both pmd and sunflow. There is a trade-off between memory usage and runtime overhead. If RV more actively removes terminated monitors, memory usage will be lower, at the cost of runtime performance. Overall, our monitor termination optimization achieves the most efficient parametric monitoring system with reasonable memory performance.

Figure 10 shows the number of triggered events, of created monitors, and of flagged monitors for each property, while JavaMOP always shows over 440% runtime overhead and Tracematches always shows over 1350%.

Also, the cases where RV does not show lower peak memory usage are within the limits of expected memory jitter. However, memory usage of RV is still higher than the memory usage of Tracematches in some cases. Tracematches has several finite automata specific memory optimizations [8], which cannot be implemented in a formalism-independent system like RV. Although Tracematches is sometimes more memory efficient, it shows prohibitive runtime overhead monitoring both pmd and sunflow. There is a trade-off between memory usage and runtime overhead. If RV more actively removes terminated monitors, memory usage will be lower, at the cost of runtime performance. Overall, our monitor termination optimization achieves the most efficient parametric monitoring system with reasonable memory performance.

Figure 10 shows the number of triggered events, of created monitors, and of flagged monitors for each property, while JavaMOP always shows over 440% runtime overhead and Tracematches always shows over 1350%.

Also, the cases where RV does not show lower peak memory usage are within the limits of expected memory jitter. However, memory usage of RV is still higher than the memory usage of Tracematches in some cases. Tracematches has several finite automata specific memory optimizations [8], which cannot be implemented in a formalism-independent system like RV. Although Tracematches is sometimes more memory efficient, it shows prohibitive runtime overhead monitoring both pmd and sunflow. There is a trade-off between memory usage and runtime overhead. If RV more actively removes terminated monitors, memory usage will be lower, at the cost of runtime performance. Overall, our monitor termination optimization achieves the most efficient parametric monitoring system with reasonable memory performance.

Figure 10 shows the number of triggered events, of created monitors, and of flagged monitors for each property, while JavaMOP always shows over 440% runtime overhead and Tracematches always shows over 1350%.
6. Conclusion
We presented an effective novel garbage collection technique for monitoring parametric properties. Previous techniques were either completely agnostic to the property to monitor, thus incurring prohibitive runtime overheads due to memory leaks, or were intrinsically dependent on particular specification formalisms, thus being hard or impossible to use in other contexts. Our technique is the first which is both formalism-generic and efficient. As extensive evaluation shows, it is in fact significantly more efficient than the existing techniques, both formalism-generic and formalism-specific.

Our results have at least two implications. On the one hand, they show that runtime monitoring of complex specifications can be used not only for testing, but also as an integral part of the deployed system in many cases. Indeed, in most practical cases the runtime overhead is negligible, so a well-designed recovery schema implemented by means of specification handlers can ensure highly dependable systems by simply not letting them go wrong at runtime. Note that the combinations program/property selected for evaluation in this paper were specifically chosen to be bad. On the other hand, our results set a solid ground for further optimizations. For example, in this paper were specifically chosen to be bad. On the other hand, our results set a solid ground for further optimizations. For example, static analyses of the program to monitor, like those in [14, 15, 21, 27], can be used to remove unnecessary instrumentation and thus not even generate many of the monitors. Similarly, staged/decentralized indexing techniques, like those in [6, 8, 17], can reduce the distance between events and their monitors and thus reduce the overhead taken to dispatch events to monitors.

References