Garbage Collection for Monitoring Parametric Properties

Dongyun Jin Patrick O'Neil Meredith Dennis Griffith Grigore Roşu

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* [30] 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.

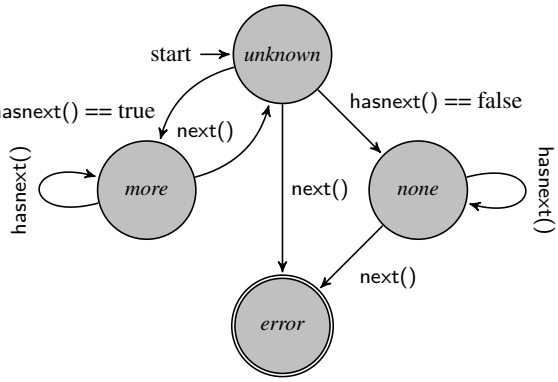

Figure 1. Typestate description for HASNEXT

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 transitions (i.e., monitored events) in brackets. The first state in the FSM description (unknown) is always considered to be the initial state of a finite state machine in the RV system. The second formalism is linear temporal logic (LTL), prefixed by the keyword ltl. The 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.

PLDI'11, June 4–8, 2011, San Jose, California, USA.

```
HasNext(Iterator i) {
  event hasnexttrue after(Iterator i) returning(boolean b) :
    call(* Iterator.hasNext()) && target(i) && condition(b) {}
  event hasnextfalse after(Iterator i) returning(boolean b) :
    call(* Iterator.hasNext()) && target(i) && condition(!b) {}
  event next before(Iterator i) :
    call(* Iterator.next()) && target(i) {}

  fsm :
    unknown [
      hasnexttrue -> more
      hasnextfalse -> none
      next -> error
    ]
    more [
      hasnexttrue -> more
      next -> unknown
    ]
    none [
      hasnextfalse -> none
      next -> error
    ]
    error [   ]
  @error {
    System.out.println("improper Iterator use found!");
  }

  ltl: [](next => (*)hasnexttrue)
  @violation {
    System.out.println("improper Iterator use found!");
  }
}
```

Figure 2. HASNEXT property in RV using FSM and LTL

```
UnsafeIter(Collection c, Iterator i) {
  event create after(Collection c) returning(Iterator i) :
    call(Iterator Collection.iterator()) && target(c) {}
  event update after(Collection c) :
    (call(* Collection.remove*(..))
    || call(* Collection.add*(..))
    || call(* Collection.clear(..))) && target(c){}
  event next before(Iterator i) :
    call(* Iterator.next()) && target(i){}

  ere : update* create next* update+ next
  @match {
    System.out.println("improper Concurrent Modification found!");
  }
}
```

Figure 3. UNSAFEITER property in RV using the ERE plugin

```
SafeLock(Lock l, Thread t){
  event acquire before(Lock l, Thread t):
    call(* Lock.acquire()) && target(l) && thread(t) {}
  event release before(Lock l, Thread t):
    call(* Lock.release()) && target(l) && thread(t) {}
  event begin before(Thread t) :
    execution(* *.*(..)) && thread(t) && !within(Lock+) {}
  event end after(Thread t) :
    execution(* *.*(..)) && thread(t) && !within(Lock+) {}

  cfg : S -> S begin S end | S acquire S release | epsilon
  @fail {    System.out.println("improper Lock use found!");  }
}
```

Figure 4. SAFELOCK property in RV using the CFG plugin

The monitored events are prefixed by the keyword event and are encoded using slightly extended AspectJ [25] 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 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 corresponding 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 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, 29], Tracematches [4, 8], JavaMOP [16, 17], PTQL [23], PQL [26], 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 the pattern that actually causes the error, there is no way to ever match the pattern if the `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 `Iterator` are garbage collected. Unfortunately, in most

realistic programs, `Collections` have much longer lifetimes than the `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 `Iterators` had been collected. The RV system, which is presented in this paper, is a commercial grade system developed by a startup company.[1] 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 coenable sets, are used at runtime to determine when a monitor instance can no longer reach a triggering state, and can thus be garbage collected. For example, in UNSAFEITER (Figure 3), the coenable sets associated to event update consist of all those subsets of events which can potentially make update a relevant event for a monitor for UNSAFEITER, 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 UNSAFEITER 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 `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, 26], 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 RV in two versions of DaCapo, from UNSAFEITER-bloat, 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.

[1] We cannot mention the company name due to double blind reviewing.

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 sets 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 reduct operation that forgets events that are unrelated to the given parameter instance.

Definition 1. *Let $\mathcal{E}$ be a finite set of (non-parametric) events, called **base events** or simply **events**. An **$\mathcal{E}$-trace**, or simply a (non-parametric) **trace** when $\mathcal{E}$ is understood or not important, is any finite sequence of events in $\mathcal{E}$, that is, an element in $\mathcal{E}^*$. If event $e \in \mathcal{E}$ appears in trace $w \in \mathcal{E}^*$ then we write $e \in w$. ϵ is the empty trace.*

For UNSAFEITER in Section 1, the set of events $\mathcal{E}$ is {create, update, next}, and a possible trace is "create next update next".

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

Consider again UNSAFEITER. 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 next", which we refer to as fail traces. Thus we pick $\mathcal{C}$ to be the set {match, fail, ?}, and define its property $P_{\text{UNSAFEITER}}\colon \mathcal{E}^* \to \mathcal{C}$ as follows: $P_{\text{UNSAFEITER}}(w) = \text{match}$ if w is in the language of the UNSAFEITER ere, $P_{\text{UNSAFEITER}}(w) = ?$ if w is a prefix of a string in the language of the ere, and $P_{\text{UNSAFEITER}}(w) = \text{fail}$ otherwise.

We next extend the above definitions to the parametric case. Let $[A \to B]$ be the set of total functions, and let $[A \rightharpoonup B]$ 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 $\mathcal{E}$ is a set of base events like in Definition 1, then let $\mathcal{E}\langle X\rangle$ be the set of corresponding **parametric events** $e\langle\theta\rangle$, where e is a base event in $\mathcal{E}$ and θ is a partial function in $[X \rightharpoonup V]$. Partial functions θ in $[X \rightharpoonup V]$ are called **parameter instances**. A **parametric trace** is a trace with events in $\mathcal{E}\langle X\rangle$, that is, a word in $\mathcal{E}\langle X\rangle^*$.*

A parametric trace for UNSAFEITER could be "update$\langle c \mapsto c_1\rangle$ update$\langle c \mapsto c_2\rangle$ create$\langle c \mapsto c_1, i \mapsto i_1\rangle$ next$\langle i \mapsto i_1\rangle$". To simplify writing we often assume the parameter set implicit, as in the following, which is the same trace: "update$\langle c_1\rangle$ update$\langle c_2\rangle$ create$\langle c_1, i_1\rangle$ next$\langle i_1\rangle$".

Definition 4. *Let X be a finite set of parameters. If $\mathcal{E}$ is a set of base events like in Definition 1, we define a **parametric event definition**, or **event definition** for short, as a function $\mathcal{D}\colon \mathcal{E} \to \mathcal{P}(X)$, where $\mathcal{P}$ is the power set, that maps each event e to a set of parameters*

$\mathcal{D}(e)$ that will be instantiated by e at runtime. $\mathcal{D}$ is extended to $\mathcal{E}^*$ as $\mathcal{D}(\epsilon) = \emptyset$ and $\mathcal{D}(ew) = \mathcal{D}(e) \cup \mathcal{D}(w)$, and to $\mathcal{P}(\mathcal{E})$ as $\mathcal{D}(\emptyset) = \emptyset$ and $\mathcal{D}(\{e\} \cup E) = \mathcal{D}(e) \cup \mathcal{D}(E)$. Parametric event $e\langle\theta\rangle$ is ***$\mathcal{D}$-consistent*** *if $\mathsf{dom}(\theta) = \mathcal{D}(e)$. Parametric trace τ is* ***$\mathcal{D}$-consistent*** *if $e\langle\theta\rangle$ is $\mathcal{D}$-consistent for each $e\langle\theta\rangle \in \tau$.*

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

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

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

$\theta \sqcup \theta'$ is also called the ***least upper bound (lub)*** *of θ and θ'. θ is* ***less informative*** *than θ', written $\theta \sqsubseteq \theta'$, if for any $x \in X$, if $\theta(x)$ is defined then $\theta'(x)$ is also defined and $\theta(x) = \theta'(x)$. $\sqcup$ is extended to $\mathcal{P}_f([X \rightharpoonup V])$ in the natural way. Here $\mathcal{P}_f$ is the finite power set.*

Definition 6. ***(Trace slicing)*** *Given parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$ and θ in $[X \rightharpoonup V]$, let the* ***θ-trace slice*** *$\tau\restriction_\theta \in \mathcal{E}^*$ be the non-parametric trace defined as:*

- $\epsilon\restriction_\theta = \epsilon$ *(recall that ϵ is the empty trace)*
- $(\tau\, e\langle\theta'\rangle)\restriction_\theta = \begin{cases} (\tau\restriction_\theta)\, e & \text{if } \theta' \sqsubseteq \theta \\ \tau\restriction_\theta & \text{otherwise} \end{cases}$

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

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

$$\Lambda X.P\colon \mathcal{E}\langle X\rangle^* \to [[X \rightharpoonup V] \to \mathcal{C}]$$

as $(\Lambda X.P)(\tau)(\theta) = P(\tau\restriction_\theta)$ for each $\tau \in \mathcal{E}\langle X\rangle^*$, $\theta \in [X \rightharpoonup V]$.

A parametric property is therefore similar to a normal property, but one partitioning parametric traces in $\mathcal{E}\langle X\rangle^*$ into verdict categories in $[[X \rightharpoonup V] \to \mathcal{C}]$, 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 $[X \rightharpoonup V]$.

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 $(S, \mathcal{E}, \mathcal{C}, \imath, \sigma, \gamma)$, where S is the set of states, $\mathcal{E}$ is the set of input events, $\mathcal{C}$ is the set of verdict categories, $\imath \in S$ is the initial state, $\sigma\colon S \times \mathcal{E} \to S$ is the transition function, and $\gamma\colon S \to \mathcal{C}$ is the verdict function. The transition function is extended to handle traces, i.e., $\sigma\colon S \times \mathcal{E}^* \to S$ where $\sigma(s, \epsilon) = s$ and $\sigma(s, ew) = \sigma(\sigma(s, e), w)$. $M = (S, \mathcal{E}, \mathcal{C}, \imath, \sigma, \gamma)$ is a* ***monitor for property*** *$P\colon \mathcal{E}^* \to \mathcal{C}$ if $\gamma(\sigma(\imath, w)) = P(w)$ for each $w \in \mathcal{E}^*$. Monitor M defines the property $P_M\colon \mathcal{E}^* \to \mathcal{C}$ with $P_M(w) = \gamma(\sigma(\imath, w))$. Monitors M and M' are equivalent iff $P_M = P_{M'}$.*

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 $M = (S, \mathcal{E}, \mathcal{C}, \imath, \sigma, \gamma)$, the* ***parametric monitor*** *$\Lambda X.M$ is the monitor $([[X \rightharpoonup V] \to S], \mathcal{E}\langle X\rangle, [[X \rightharpoonup V] \to \mathcal{C}], \lambda\theta.\imath, \Lambda X.\sigma, \Lambda X.\gamma)$, with*

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

defined as

$$(\Lambda X.\sigma)(\delta, e\langle\theta'\rangle)(\theta) = \begin{cases} \sigma(\delta(\theta), e) & \text{if } \theta' \sqsubseteq \theta \\ \delta(\theta) & \text{otherwise} \end{cases}$$
$$(\Lambda X.\gamma)(\delta)(\theta) = \gamma(\delta(\theta))$$

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

Therefore, a parametric monitor $\Lambda X.M$ maintains a state $\delta(\theta)$ of M for each parameter instance θ, takes parametric events as input, and outputs categories indexed by parameter instances (one 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 one trace slice. The rule for $\Lambda X.\sigma$ can be read as stating that when an event with parameter instance θ' is evaluated, it updates the state for all monitor instances more informative than the instance for θ', and the instance for θ' itself, leaving all other monitor instances untouched. The rule for $\Lambda X.\gamma$ simply states that γ is applied to a state, as normal, but the state is found by looking up the state of the monitor instance for θ. 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, MONITOR(M) yields a monitor that is equivalent to $\Lambda X.M$, that is, a monitor for $\Lambda X.P$. The functions $[[X \rightharpoonup V] \to S]$ and $[[X \rightharpoonup V] \to \mathcal{C}]$ of $\Lambda X.M$ are encoded by MONITOR(M) as tables Δ and Γ with entries indexed by parameter instances in $[X \rightharpoonup V]$ and with contents states in S and verdict categories in $\mathcal{C}$, respectively. Such tables will have finite entries because each event e binds only a finite number of parameters defined by $\mathcal{D}(e)$.

The monitoring algorithm first clears Δ, which contains the monitor state for each parameter instance, then assigns $\imath$, the initial state, to $\Delta(\bot)$. Θ, which contains all known parameter instances, is initialized to contain only the empty partial function $\bot$. For

```
Algorithm MONITOR(M = (S, E, C, ı, σ, γ))
function main(τ)
1  Δ ← ⊥; Δ(⊥) ← ı; Θ ← {⊥}
2  foreach e⟨θ⟩ in order in τ do
3  : foreach θ' ∈ {θ} ⊔ Θ do
4  : : Δ(θ') ← σ(Δ(max{θ'' ∈ Θ | θ'' ⊑ θ'}), e)
5  : : Γ(θ') ← γ(Δ(θ'))
6  : endfor
7  : Θ ← {⊥, θ} ⊔ Θ
8  endfor
```

Figure 5. Monitoring Algorithm

each event $e\langle\theta\rangle$ that arrives during program execution (line 2), MONITOR(M) generates every compatible parameter instance by combining θ with all the previously known compatible parameter instances (line 3). It then updates the state of every one of these compatible parameter instances (θ') with the state, transitioned by event e, of the "monitor instance" corresponding to the "largest" parameter instance less than or equal to θ' (line 4). At the same time we also calculate the verdict category corresponding to that monitor instance and store it in table Γ (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 θ'. After the algorithm completes, Γ 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 $\mathcal{G}$. Indeed, as seen in Section 5, the RV garbage collection technique flags 1.8 million of these monitor instances as unnecessary. Chen et al. [19] proposed a formalism-independent method, called "ENABLE sets", to avoid needlessly *creating* monitors that will never trigger. Here we show how a dual method can be derived to avoid needlessly *retaining* monitors that will never trigger. Computing the coenable sets is expected to be a quick static 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 $w \in \mathcal{E}^*$ and $e, e' \in w$, we let $e \rightsquigarrow_w e'$ denote that e' occurs after e in w. Let* $\text{COENABLE}_w(e) = \{e' \mid e \rightsquigarrow_w e'\}$ *be the* ***trace coenable set*** *of e. Given property $P\colon \mathcal{E}^* \to \mathcal{C}$ and a subset of verdict categories of interest (or goal) $\mathcal{G} \subseteq \mathcal{C}$, the* ***property coenable set*** *is defined as the map* $\text{COENABLE}_{P,\mathcal{G}}\colon \mathcal{E} \to \mathcal{P}(\mathcal{P}(\mathcal{E}))$ *where* $\text{COENABLE}_{P,\mathcal{G}}(e) = \{\text{COENABLE}_w(e) \mid w \in \mathcal{E}^* \text{ s.t. } P(w) \in \mathcal{G}, e \in w, \text{ COENABLE}_w(e) \neq \emptyset\}$ *for each $e \in \mathcal{E}$.*

Intuitively, if event e is encountered during monitoring, but none of the event sets of $\text{COENABLE}_{P,\mathcal{G}}(e)$ are possible in the future, it is impossible to reach any verdict category in $\mathcal{G}$, so a monitor for P observing e will never trigger. We drop all $\emptyset$s from $\text{COENABLE}_{P,\mathcal{G}}$ because they can cause monitor instances to be retained that are unnecessary. An $\emptyset$ in $\text{COENABLE}_{P,\mathcal{G}}(e)$ means that the trace suffix consisting of only the event e can lead to a category in $\mathcal{G}$ for some trace prefix. However, our interest is in the ability to reach $\mathcal{G}$ again in the future. If there is a trace suffix that can lead to a state in $\mathcal{G}$ from e, then its events will be added to $\text{COENABLE}_{P,\mathcal{G}}(e)$. If there is no trace suffix that can lead back to a state in $\mathcal{G}$, 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 $(S, \mathcal{E}, \mathcal{C}, \imath, \sigma, \gamma)$ where $\mathcal{E}$ is a finite alphabet, S is a finite set of states, $\imath \in S$ is the initial state, $\sigma\colon S \times \mathcal{E} \rightharpoonup S$ a partial transition function, $\mathcal{C}$ a set of verdict categories, and $\gamma\colon S \to \mathcal{C}$ the verdict function. The property monitored by an FSM classifies a trace w into $\gamma(\sigma(\imath, w))$, where σ is extended to strings in the natural way, and fail if $\sigma(\imath, w)$ is undefined.

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

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

$$\text{COENABLE}_{P,\mathcal{G}}(e) = \bigcup_{\sigma(s,e)=s'} \text{SEEABLE}(s')$$

We can use the equations above to generate coenable sets for our example from Figure 3, one need simply generate a finite state machine from the property's ERE. For $P =$ UNSAFEITER and $\mathcal{G} = \{\mathsf{match}\}$, the $\text{COENABLE}_{P,\mathcal{G}}$ sets are:

$$\text{COENABLE}_{P,\mathcal{G}}(\mathsf{create}) = \{\{\mathsf{next}, \mathsf{update}\}\}$$

$$\text{COENABLE}_{P,\mathcal{G}}(\mathsf{update}) = \left\{\begin{matrix}\{\mathsf{next}\}, \{\mathsf{next}, \mathsf{update}\}, \\ \{\mathsf{next},\ \mathsf{create},\ \mathsf{update}\}\end{matrix}\right\}$$

$$\text{COENABLE}_{P,\mathcal{G}}(\mathsf{next}) = \{\{\mathsf{next}, \mathsf{update}\}\}$$

Note that if we did not remove $\emptyset$s, $\text{COENABLE}_{P,\mathcal{G}}(\mathsf{next})$ 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 $\mathcal{G}$ to still 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 $(N, \mathcal{E}, \mathtt{S}, \Pi)$ where N is a finite set of nonterminals, $\mathcal{E}$ is a finite set of terminals, $\mathtt{S} \in N$ is the initial nonterminal, and Π is a set of productions of the form $\mathtt{A} \to \beta$ where $\mathtt{A} \in N$ and $\beta \in (N \cup \mathcal{E})^*$. The monitor for a CFG classifies traces that are in the language of the grammar into the verdict category match.

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

$$\mathrm{G}(\epsilon) = \{\emptyset\} \quad \mathrm{G}(e) = \{\{e\}\} \quad \mathrm{G}(\mathtt{A}) = \textstyle\bigcup_{\mathtt{A}\to\beta} \mathrm{G}(\beta)$$

$$\mathrm{G}(\beta_1\beta_2) = \{T_1 \cup T_2 \mid T_1 \in \mathrm{G}(\beta_1), T_2 \in \mathrm{G}(\beta_2)\}$$

$$C(x) = \left\{ T_1 \cup T_2 \;\middle|\; \begin{matrix}\mathtt{A} \to \beta_1 x \beta_2, \\ T_1 \in C(\mathtt{A}), T_2 \in \mathrm{G}(\beta_2)\end{matrix} \right\}$$

$$\text{COENABLE}_{P,\{\mathsf{match}\}}(e) = C(e)$$

Informally, $\mathrm{G}(\mathtt{A})$ is the set of events generated by the CFG, if the symbol $\mathtt{A}$ were used as the initial nonterminal of the CFG. The equation $\mathrm{G}(\beta_1\beta_2) = \{T_1 \cup T_2 \mid T_1 \in \mathrm{G}(\beta_1), T_2 \in \mathrm{G}(\beta_2)\}$ generalizes this notion to entire traces of symbols (where symbols are either events or non-terminals). C is the coenable sets function generalized to traces that include both non-terminals and events. For a production, $\mathtt{A} \to \beta_1 \mathtt{B} \beta_2$, $C(\mathtt{B})$ needs to cope with the fact that $\mathtt{A}$ has its own coenable sets. Thus its definition unions possible coenable sets of $\mathtt{A}$ with the sets of symbols that are generated by β_2. The rest of RV only needs to know coenable sets for events so coenables is just the restriction of C to events.

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

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

sets are:

$$\text{COENABLE}^X_{P,\mathcal{G}}(\mathsf{create}) = \{\{c, i\}\}$$
$$\text{COENABLE}^X_{P,\mathcal{G}}(\mathsf{update}) = \{\{i\}, \{c, i\}\}$$
$$\text{COENABLE}^X_{P,\mathcal{G}}(\mathsf{next}) = \{\{c, i\}\}$$

Now with the $\text{COENABLE}^X_{P,\mathcal{G}}$ 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 $\text{COENABLE}^X_{P,\mathcal{G}}(e)$ for a trace wew' 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 $wew' \in \mathcal{E}^*$. If for each $Y \in \text{COENABLE}^X_{P,\mathcal{G}}(e)$ there exists some $y \in Y$ such that $y \notin \mathcal{D}(w')$ then $P(wew') \notin \mathcal{G}$.*

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

Discussion The $\text{COENABLE}^X_{P,\mathcal{G}}$ 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 $x \in \mathcal{D}(e)$ can only occur in a trace-slice $\tau\restriction_\theta$ if $\theta(x)$ is still alive in the system. If $\theta(x)$ has been garbage collected, there is no way for any e with $x \in \mathcal{D}(e)$ to occur in trace slice for θ. This is precisely how monitoring arrives in the situation presented in Theorem 1, where all possible suffixes w' of the trace slice wew' do not contain at least one parameter in each set of the $\text{COENABLE}^X_{P,\mathcal{G}}(e)$, and it becomes impossible to reach a verdict category in $\mathcal{G}$. Clearly, if it is impossible for the θ trace slice to ever reach a verdict category in $\mathcal{G}$, there is no reason to keep the monitor instance for θ.

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 coenables 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 coenables 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 coenables technique reclaims 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).

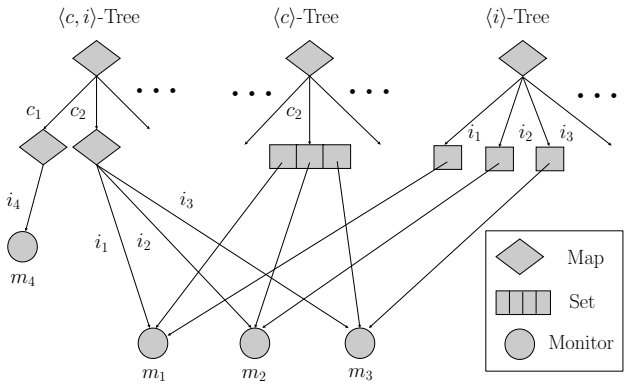

Figure 6. Indexing trees for UNSAFEITER

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 Δ and Γ 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 $e\langle\theta\rangle$ is processed, all monitor instances corresponding to parameter instances more informative than θ 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 $\mathsf{update}\langle c_2\rangle$ for UNSAFEITER (Figure 3) requires that we update the monitor instances that are more informative than $\langle c_2\rangle$. Thus we lookup $\langle c_2\rangle$ in the $\langle c\rangle$-tree of Figure 6 to find the set of monitor instances more informative than $\langle c_2\rangle$ 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 $\langle c\rangle$-tree, an $\langle i\rangle$-tree, and a $\langle c, i\rangle$-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

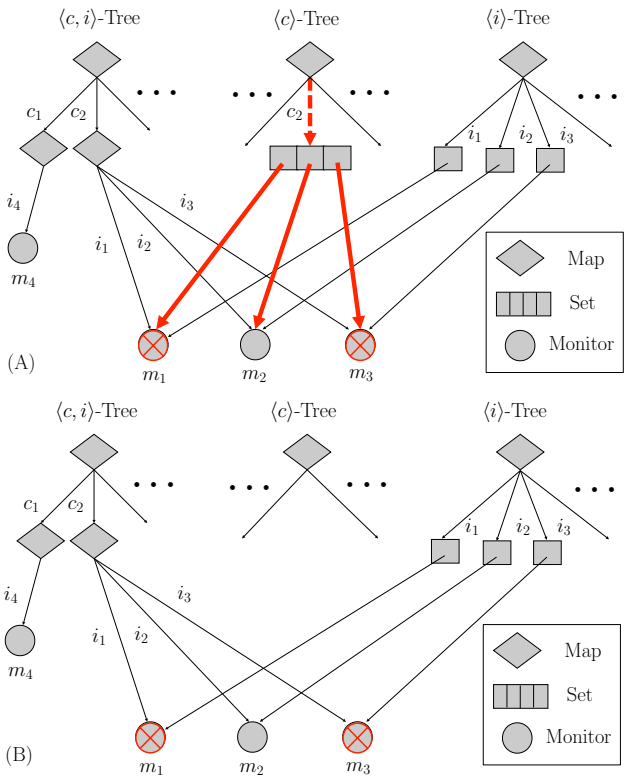

Figure 7. (A) Notifying monitors for garbage collected $\langle c_2 \rangle$ in the $\langle c \rangle$-tree. (B) Cleaning up the broken mapping in the $\langle c \rangle$-tree

a monitor instance for $\langle c_1, i_1 \rangle$ and i_1 is garbage collected, even 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 informs all the relevant monitor instances contained within itself. Then, the monitor instance decides if it can still possibly reach a verdict category in $\mathcal{G}$ 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.

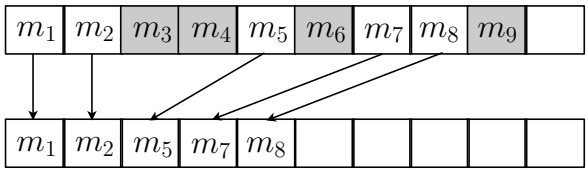

Figure 8. A compaction in `RVSet` when some monitor instances are collectable

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 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 $\text{COENABLE}^X_{P,\mathcal{G}}(e)$, when this notification takes place. The monitor instance need simply check if all the parameter objects of any set in $\text{COENABLE}^X_{P,\mathcal{G}}(e)$ are alive. RV statically translates $\text{COENABLE}^X_{P,\mathcal{G}}(e)$ to a minimized boolean formula to make this check as efficient as possible:

$$\text{ALIVENESS}(e) = \bigvee_{S \in \text{COENABLE}^X_{P,\mathcal{G}}(e)} \left(\bigwedge_{x \in S} \mathit{live}_x \right)$$

where 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 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 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.

5. Evaluation of the RV System

We evaluate our formalism-independent garbage collection for parametric monitoring implemented into RV. Also, we compare the

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

Figure 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)

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 19194% runtime overhead (i.e., 192 times slower) and uses 7.7MB of heap memory in Tracematches, and causes 569% runtime overhead and uses 147MB in JavaMOP, while the original program uses only 4.9MB. Also, although the DaCapo 9.12 provides jython, Tracematches cannot instrument jython due to an error. 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 $\pm 3\%$. We also looked into other benchmarks including Java Grande [28] 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, 27].

- 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);
- UNSAFEMAPITER: Do not update a Map when using the Iterator interface to iterate its values or its keys;
- UNSAFESYNCCOLL: If a Collection is synchronized, then its iterator also should be accessed synchronously;
- UNSAFESYNCMAP: If a Collection is synchronized, then its iterators on values and keys also should be accessed in a synchronized manner.

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

	HASNEXT				UNSAFEITER				UNSAFEMAPITER				UNSAFESYNCCOLL				UNSAFESYNCMAP			
	E	M	FM	CM	E	M	FM	CM	E	M	FM	CM	E	M	FM	CM	E	M	FM	CM
bloat	156M	1.9M	1.9M	1.8M	81M	1.9M	1.8M	1.6M	73M	3.6M	44K	3.5M	143M	4.1M	0	3.7M	161M	3.4M	0	3.4M
jython	106	50	47	26	179K	50	38	38	179K	101K	94	101K	156	100	0	83	256	150	0	122
avrora	1.5M	909K	850K	765K	1.4M	909K	860K	808K	1.3M	1.2M	18	1.2M	2.4M	1.8M	0	1.7M	1.5M	909K	0	904K
batik	49K	24K	21K	21K	125K	24K	21K	10K	55K	33K	140	27K	73K	50K	0	34K	50K	26K	0	26K
eclipse	226K	7.6K	5.3K	2.9K	119K	6.6K	5.1K	2.6K	113K	22K	2.2K	7.8K	233K	15K	0	7.5K	241K	18K	0	9.2K
fop	1.0M	184K	74K	151K	709K	7.7K	7.2K	1.8K	499K	177K	67	160K	1.2M	239K	0	217K	1.2M	231K	0	213K
h2	27M	6.5M	6.0M	5.6M	12M	3.7K	3.3K	1.3K	12M	6.6M	9	6.5M	27M	6.5M	0	6.5M	27M	6.5M	0	6.5M
luindex	371	66	40	2	4.4K	65	39	0	378	183	2	59	436	132	0	30	472	125	0	25
lusearch	1.4K	131	196	114	748K	130	210	18	20K	944	338	1.4K	1.7K	262	0	402	1.8K	263	0	158
pmd	8.3M	789K	694K	571K	6.4M	551K	473K	382K	4.3M	1.3M	110K	1.1M	8.8M	1.5M	0	1.3M	8.6M	1.1M	0	999K
sunflow	2.7M	101K	101K	100K	1.3M	2	0	0	1.3M	83K	0	83K	2.7M	101K	0	101K	2.7M	101K	0	101K
tomcat	25	6	0	0	132	4	0	0	68	26	0	0	29	10	0	0	33	12	0	0
tradebeans	11	3	0	0	31	2	0	0	29	13	0	0	13	5	0	0	15	6	0	0
tradesoap	11	3	0	0	31	2	0	0	29	13	0	0	13	5	0	0	15	6	0	0
xalan	11	3	0	0	8.9K	2	0	0	119K	20K	0	20K	13	5	0	0	15	6	0	0

Figure 10. Monitoring statistics: number of events (E), number of created monitors (M), number of flagged monitors (FM), number of collected monitors (CM).

for performance reasons or structural limitations. We have tested several non-`Iterator` based properties: HASHSET, SAFEENUM, SAFEFILE, and SAFEFILEWRITER [13, 14, 19, 27]. None of these properties produce overheads above 5% in any of the DaCapo benchmarks, thus their results are not presented in this paper in detail.

5.1.1 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 $\langle c, i \rangle$-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 have `null` referents. The value can be either a monitor instance, a set, or a lower level map. If the value is a flagged monitor instance or an empty data structure, it removes the mapping. If it is a set, it must be checked for internal monitor instances that have been flagged for removal. When a set is checked for unnecessary monitor instances, all of the instances are collected, and the remaining necessary monitor instances are compacted in one pass, as can be seen in Figure 8.

5.2 Results and Discussions

Figures 9 and 10 summarize the results of the evaluation. Note that the structure of the DaCapo 9.12 allows us to instrument all of the benchmarks plus all supplementary libraries that the benchmarks use, which was not possible for DaCapo 2006-10. Therefore, fop and pmd show higher overheads than the benchmarks using DaCapo 2006-10 from [19]. While other benchmarks show overheads less than 80% in JavaMOP, bloat, avrora, and pmd show prohibitive overhead in both runtime and memory performance. This is because they generate many iterators and all properties in this evaluation are intended to monitor iterators. For example, bloat creates 1,625,770 collections and 941,466 iterators in total while 19,605 iterators coexist at the same time at peak, in an execution. avrora and pmd also create many collections and iterators. Also, they call hasNext() 78,451,585 times, 1,158,152 times and 4,670,555 times and next() 77,666,243 times, 352,697 times and 3,607,164 times, respectively. Therefore, we mainly discuss those three examples in this section, although RV shows improvements for other examples as well.

Figure 9 (A) shows the percent runtime overhead of Tracematches, JavaMOP, and RV. Overall, RV averages two times less runtime overhead than JavaMOP and orders of magnitude less runtime overhead than Tracematches (recall that these are the most optimized runtime verification systems). With bloat, RV shows less than 260% runtime overhead for each property, while JavaMOP always shows over 440% runtime overhead and Tracematches always shows over 1350% for completed runs and *crashed* for UNSAFEMAPITER. With avrora, on average, RV shows 62% runtime overhead, while JavaMOP shows 139% runtime overhead and Tracematches shows 203% and hangs for UNSAFEMAPITER. With pmd, on average, RV shows 94% runtime overhead, while JavaMOP shows 231% runtime overhead and Tracematches shows 1139% and hangs for UNSAFEMAPITER and UNSAFESYNCMAP.

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

Figure 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 bloat and pmd. There is a trade-off between memory usage and runtime overhead. If RV more actively removes terminated monitors, memory usage will be lower, at the cost of runtime performance. Overall, 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, of monitors flagged as unnecessary by the coenable set technique, and of monitors collected by the JVM. Among the DaCapo examples, bloat, avrora, h2, pmd and sunflow generated a very large number of events (more than a million) in all properties, resulting in millions of monitors created in most cases. h2 does not exhibit large overhead because monitor instances in h2 have shorter lifetimes, therefore the created monitor instances are not used heavily like in bloat. sunflow has millions of events but does

not create as many monitor instances as as other benchmarks. When monitoring the HASNEXT and UNSAFEITER properties, the coenable sets technique effectively flagged monitors as unnecessary and most were collected by the JVM.

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, static analyses of the program to monitor, like those in [14, 15, 21, 26], 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

[1] JavaMOP. http://javamop.com.

[2] SPECjvm 2008. http://www.spec.org/jvm2008/.

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

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

[5] M. Arnold, M. Vechev, and E. Yahav. Qvm: an efficient runtime for detecting defects in deployed systems. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'08)*, pages 143–162. ACM, 2008.

[6] P. Avgustinov and C. Church. *Trace Monitoring with Free Variables*. PhD thesis, Oxford University, 2009.

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

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

[9] H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for runtime monitoring: from EAGLE to RULER. *J. Logic Computation*, November 2008.

[10] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'06)*, pages 169–190. ACM, 2006.

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

[12] E. Bodden and V. Stolz. Tracechecks: Defining semantic interfaces with temporal logic. In *Software Composition*, pages 147–162, 2006.

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

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

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

[16] F. Chen and G. Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 546–550. Springer, 2005.

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

[18] F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 246–261. Springer, 2009.

[19] F. Chen, P. Meredith, D. Jin, and G. Roşu. Efficient formalism-independent monitoring of parametric properties. In *Automated Software Engineering (ASE'09)*, pages 383–394. IEEE, 2009.

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

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

[22] U. Erlingsson and F. B. Schneider. Irm enforcement of java stack inspection. In *Symposium on Security and Privacy (SP'00)*, pages 246–. IEEE, 2000.

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

[24] K. W. Hamlen and M. Jones. Aspect-oriented in-lined reference monitors. In *Programming languages and analysis for security (PLAS'08)*, pages 11–20. ACM, 2008.

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

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

[27] P. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. *J. Automated Software Engineering*, 17(2):149–180, June 2010.

[28] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *Supercomputing (SC'01)*, pages 8–8. ACM, 2001.

[29] V. Stolz and E. Bodden. Temporal Assertions using AspectJ. In *Runtime Verification (RV'05)*, volume 144 of *ENTCS*, pages 109–124. Elsevier, 2005.

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