Scalable Parametric Runtime Monitoring

Dongyun Jin Patrick O'Neil Meredith Grigore Roşu

Department of Computer Science
University of Illinois at Urbana–Champaign
Urbana, IL, U.S.A.
{djin3, pmeredit, grosu}@cs.illinois.edu

Abstract

Runtime monitoring is an effective means to improve the reliability of systems. In recent years, parametric monitoring, which is highly suitable for object-oriented systems, has gained significant traction. Previous work on the performance of parametric runtime monitoring has focused on the performance of monitoring only one specification at a time. A realistic system, however, has numerous properties that need to be monitored simultaneously. This paper introduces scalable techniques to improve the performance of one of the fastest parametric monitoring systems, JavaMOP, in the presence of multiple simultaneous properties, resulting in average runtime overheads that are less than the summation of the overheads of the properties run in isolation. An extensive evaluation shows that these techniques, which were derived following a thorough investigation and analysis of the current bottlenecks in JavaMOP, improve its runtime performance in the presence of multiple properties by up to two times and the memory usage by 34%.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications; D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing/Debugging

General Terms Languages, Performance, Reliability, Verification

Keywords scalability, parametric monitoring, runtime verification, runtime monitoring, testing, debugging, aspect-oriented programming

1. Introduction

Runtime monitoring is an effective technique to increase software reliability, by enabling more effective testing, debugging, and recovery from incorrect program behavior.

Parametric properties are properties that describe behaviors of objects (parameters), which a program should conform during its execution. They can describe use protocols for classes, pre-conditions for using classes, prohibited activities, and so fourth. Typestates [30] are a similar concept, but only allow one single parameter. Parametric properties in general can describe properties about any number of parameters. For example, Figure 1 shows the Map_UnsafeIterator specification from [26], which formalizes the parametric property concerning `Map`, `Collection`, and `Iterator` parameters, that "a map should not be updated while using the iterator interface to iterate over its keys or values." This specification defines five parametric events with the corresponding AspectJ pointcuts. The property is formalized using an extended regular expression (ERE), as specified by the ere keyword. If a program behavior matches this pattern, violating the property from the Java API documentation, the defined handler containing the user-defined Java code will be executed; here we simply print out an error message in the handler. Handler can be any code, from logging to recovery.

Many runtime properties can be enforced with parametric monitoring. Parametric specifications are especially effective for formalizing properties that arise in object-oriented programming. Several parametric monitoring systems such as Hawk/Eagle [16], J-Lo [8, 9, 29], JavaMaC [25], JavaMOP [13–15], JPaX [23], Pal [12], PoET [19], PQL [27], PTQL [21], RuleR [6], QVM [4], SpoX [22], Temporal-Rover [17], and Tracematches [3, 5] have been proposed in recent years. Among those systems, JavaMOP is the only formalism-independent parametric monitoring system.

Designing and developing a system which can *efficiently* monitor parametric properties is not trivial. Many advanced monitoring systems often show prohibitive overhead for complex specifications or for complex applications. This is because the parameters are dynamically bound to objects at runtime, resulting in a potentially unlimited number of parameter bindings [24]. Since the property needs to be

```
Map_UnsafeIterator(Map m, Collection c, Iterator i) {
  creation event getset after(Map m) returning(Collection c) :
    (call(Set Map+.keySet()) || call(Collection Map+.values()))
    && target(m) {}

  event getiter after(Collection c) returning(Iterator i) :
    call(Iterator Iterable+.iterator()) && target(c) {}

  event modifyMap before(Map m) :
    (call(* Map+.clear*(..)) || call(* Map+.put*(..))
    || call(* Map+.remove(..))) && target(m) {}

  event modifyCol before(Collection c) :
      (call(* Collection+.clear(..))
      || call(* Collection+.offer*(..))
      || call(* Collection+.pop(..))
      || call(* Collection+.push(..))
      || call(* Collection+.remove*(..))
      || call(* Collection+.retain*(..))) && target(c) {}

  event useiter before(Iterator i) :
    (call(* Iterator.hasNext(..))
    || call(* Iterator.next(..))) && target(i) {}

  ere : getset (modifyMap | modifyCol)* getiter useiter*
        (modifyMap | modifyCol)+ useiter

  @match {
    System.err.println("a violation detected!");
  }
}
```

Figure 1. Map_UnsafeIterator specification in JavaMOP

checked for each parameter binding individually, the runtime and memory overhead of monitoring a parametric property can be arbitrarily large. There has been a significant amount of research on improving the runtime and the memory performance of parametric monitoring [5, 11, 24, 28]. Thanks to these efforts, parametric monitoring has become relatively practical: in most cases the runtime/memory overhead is not noticeable; and it generally costs less than 15% runtime overhead to monitor even complex object systems and/or parametric properties; and even in extreme cases the runtime overhead is reasonable [24]. Static optimization techniques, such as [11], can improve performance even further (but we do not investigate static analyses in this paper). For the rest of this paper, whenever we say *monitoring* we mean *parametric monitoring*.

To the best of our knowledge, all earlier efforts on parametric monitoring have been focusing on better performance when monitoring a single specification. In reality, it is quite likely to have many specifications for a given program. A natural question then is: can we do better than the sum of the parts? That is, can we monitor multiple specifications at the same time with less overhead than the sum of the overheads of each individual property? Theoretically, if all specifications are independent from each other without any overlap in declared events or parameter types, there is no way to monitor them more efficiently. However, in practice, there are likely multiple specifications on the same class, often sharing some events and parameter types. Among 137 specifications from [26], only 42 specifications are totally independent from all the other specifications.

In this paper, we present scalable parametric monitoring techniques for monitoring multiple simultaneous specifications more efficiently in the presence of some overlaps between specifications. The main idea of the scalable techniques is to share resources for monitoring between specifications, reducing the memory usage and utilizing the caches more often. Since our scalable techniques are formalism-independent and address general issues in the indexing tree technique, they can be applied to other parametric monitoring systems that use similar indexing tree structures. Also, they are orthogonal to other optimization techniques like static optimization [10, 11, 18, 27], which reduce runtime and memory overhead significantly. However, we deliberately disabled static optimizations in this paper to measure the effectiveness of our scalable techniques properly.

For the evaluation of our work on scalability, we use the 137 specifications from [26]. These specifications are based on the Java 6 API documentation concerning three main packages: java.io, java.lang, and java.util. Since there is no other parametric monitoring tool which is capable of practically monitoring the 137 specifications simultaneously, we compare our work on scalability to the previous version of JavaMOP. The average runtime overhead of the scalable JavaMOP run on version 9.12 of the DaCapo [7] benchmark suite for the 137 simultaneous specifications is 147%, which is almost half of the 262% overhead that the previous JavaMOP shows. Moreover, this runtime overhead is smaller than the sum of the overheads from monitoring them individually, which is 178% on average, while the previous version of JavaMOP shows more overhead when running all properties together than the sum of the overheads when run individually, which is 243% on average.

The rest of this paper is structured as follows: Section 2 provides background on parametric monitoring as used in the previous version of JavaMOP; Section 3 presents a thorough profiling of current runtime overheads from monitoring, and discusses the main current bottlenecks in monitoring; Section 4 discusses our scalable parametric monitoring techniques in detail; Section 5 presents our evaluation results for the 137 specifications; Section 6 discusses some ineffectual approaches that we have tried; and Section 7 concludes.

Contributions This paper's contributions are as follows:

- *Thorough Profiling* on overhead from monitoring;
- *Scalable Monitoring Techniques* that reduce runtime and memory overhead significantly when monitoring multiple specifications simultaneously;
- *Large Scale Evaluation* that monitors 137 parametric specifications simultaneously for the first time.

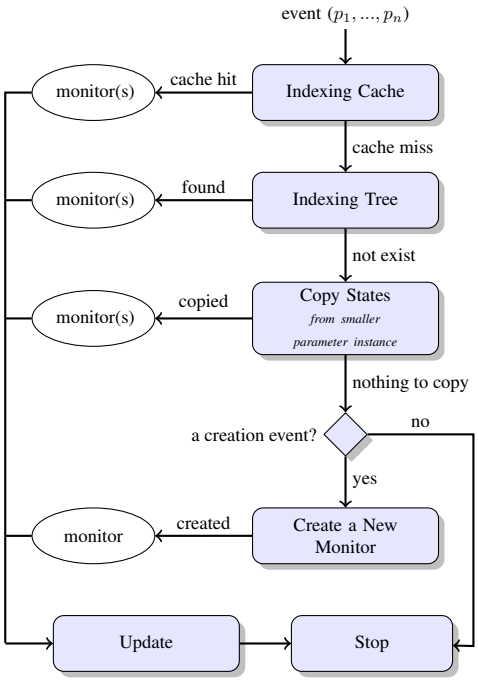

Figure 2. Overview of the previous monitoring mechanism

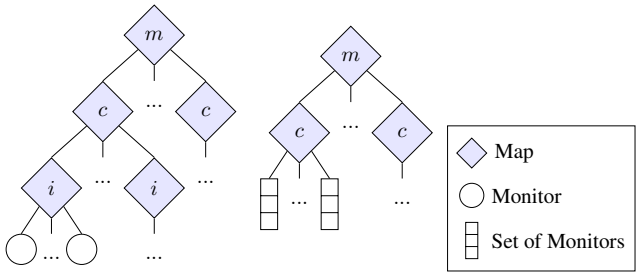

Figure 3. Some indexing trees for Map_UnsafeIterator

2. Background

To explain scalable parametric monitoring techniques, we first introduce some background on the previous parametric monitoring algorithm, its implementation, and data structures for it. A more in-depth discussion of these details can be found in [24, 28]. In *parametric monitoring*, a specification monitors a program execution trace consisting of *parametric events*. The specification must hold for each parameter instance, which is a partial mapping from parameters to objects. A parametric event e with parameter instance p is denoted as $e\langle p \rangle$. For example, the Map_UnsafeIterator specification (Figure 1) should hold for all combinations of a `Map` instance, a `Collection` instance, and an `Iterator` instance that are related by specification events.

We slice the program execution trace for each parameter instance so that a monitor for each parameter instance can forget about parameters and focus on the property. In this way, a monitor is independent from parameters, resulting in a formalism-independent parametric monitoring system. In this paper, we omit the detailed algorithm of the parametric monitoring, but summarize it in Figure 2 since the detailed algorithm is not required in explaining our work; the interested readers may refer to [28]. Upon an event, it retrieves the monitor(s) from the indexing tree. If there is no monitor that can accept a given parametric event, $e\langle p \rangle$, either an existing monitor from a subset of p is copied, or if no such monitor exists and e is a creation event, a new monitor is created. For better performance, it uses an indexing cache that stores the previously accessed monitor(s).

The indexing tree is an efficient means to locate the monitors for a given parameter instance. The indexing tree is implemented as a multi-level map that, at each level, indexes each parameter object of the parameter instance. For example, Figure 3 shows two indexing trees out of six in the Map_UnsafeIterator specification. The indexing tree for $\langle$`Map`, `Collection`, `Iterator`$\rangle$ (left tree) is a 3-level map. With a map, a collection, and an iterator, we can retrieve the related monitor. The indexing tree for $\langle$`Map`, `Collection`$\rangle$ (right tree) is a 2-level map. For a map and a collection, this indexing tree returns a set of monitors, because there can be multiple monitors for the given map and collection (one monitor for each iterator).

If an indexing tree stores all parameter objects directly, it will block them from being garbage collected, leading to a memory leak. Instead of storing parameter objects directly, the indexing tree uses the `WeakReference` class from the Java API. `WeakReference` allows a reference to an object that will not disallow garbage collection for said object. When the object is garbage collected, the JVM changes the referent field of all weak references referring to it to `null`. In this way, parameter objects can be garbage collected without any interference from monitoring.

Mappings in the indexing tree can be broken when parameter objects are garbage collected and their weak references point to `null`. The Java API provides a way to queue weak references of garbage collected objects into a `ReferenceQueue` object. By using this feature, broken mappings can be easily removed from the indexing tree. However, using this feature slows down the system significantly, because queuing weak references involves synchronization. While other general data structures like the Apache Commons Collections Library [20], use this feature, our implementation of the indexing tree does not use it for the performance reasons. Instead of using `ReferenceQueue`, we iterate through mappings and remove broken ones. Surprisingly, iterating through mappings is significantly faster than using the queuing feature from the Java API. This self-cleaning feature of our indexing tree also allows for efficient garbage collection of unnecessary monitors [24]; when we iterate through the mappings, we simply check whether any of the monitors have become unnecessary.

Overhead Fraction	Method Name
355%	Original Program
281%	MOPSet.event
205%	MOPMap.cleanup
130%	System.identityHashCode
69%	MOPMap.get
67%	MOPSet.size
51%	MOPMap.endObject
28%	Aspect Code
27%	MOPMap.full_cleanup
22%	MOPSet.endObject

Table 1. Overhead distribution when monitoring bloat (total overhead: 1330%)

Overhead Fraction	Method Name
479%	Original Program
90%	MOPSet.event
56%	MOPMap.cleanup
28%	System.identityHashCode
25%	MOPSet.size
13%	MOPMap.get
8%	MOPMap.full_cleanup
7%	MOPMap.endObject
6%	WeakReference ⟨init⟩
5%	MOPSet.endObject

Table 2. Overhead distribution when monitoring pmd (total overhead: 831%)

3. Overhead Analysis

In this section, we analyze the overhead of monitoring to find the main bottlenecks in monitoring. For this analysis, we have selected 9 specifications[1] that have caused the most overhead in previous evaluations. We run the specifications on the bloat and pmd benchmarks because they have shown the largest overheads among the benchmarks in our evaluation (Section 5). We use the same system settings from the evaluation, and HPROF, the Heap/CPU profiling tool included in the Sun JDK [2] is used to obtain performance statistics. There are two modes for CPU usage analysis in HPROF: the CPU Usage Times Profile and the CPU Usage Sampling Profile. The CPU Usage Times Profile adds a considerable amount of overhead, obstructing the analysis of the actual bottlenecks. Moreover, we do not need to know the exact time distribution to figure out where bottlenecks occur. The CPU Usage Sampling Profile, which causes less performance degradation, is good enough for this analysis. Since the CPU Usage Sampling Profile does not combine the results for the same method of different object instances, we manually combine them and categorize.

Tables 1 and 2 summarize the profiling results for monitoring bloat and pmd. The results for bloat show total overhead of 1330%; that is 1430% total execution time compared to the original non-monitored bloat. In the same way, monitoring pmd shows a total overhead of 831%. Because profiling can change the program behavior, numbers may contain errors, so they should be considered as rough estimations.

The MOPSet.event entry in Table 1 shows the overhead spent updating monitor states when events occur. This component is formulated from the property of the specification, and is already optimized well. MOPMap.cleanup and MOPMap.full_cleanup remove mappings of garbage collected parameter objects and monitors. The difference is whether it partially or fully scans the map. These cleanup methods are well tuned so that they are unlikely to be improved significantly. The methods MOPMap.endObject and MOPSet.endObject propagate information about garbage collected parameters. They consist of simple statements and have already been thoroughly optimized [24].

[1] Map_UnsafeIterator, Collection_UnsafeIterator, Iterator_HasNext, Collections_SynchronizedCollection, NavigableMap_Modification, Collections_SynchronizedMap, Iterator_RemoveOnce, List_UnsynchronizedSubList, Collections_SortBeforeBinarySearch

Description	Peak Memory Usage	Young Garbage Collection Time	Full Garbage Collection Time
Original bloat	5MB	6%	2%
Original pmd	21MB	7%	8%
Monitoring bloat (out of 1330% overhead)	970MB	278%	258%
Monitoring pmd (out of 831% overhead)	603MB	172%	181%

Table 3. Memory usage analysis

System.identityHashCode is the system default hashing function provided in the Java API, which is based on reference identity instead of the equals method provided by classes. It returns the same hash code for objects a and b if a == b, and tries to return different codes otherwise, but uniqueness is not guaranteed. Although this is just one of several statements in the MOPMap.get method that retrieves monitor(s) for a parameter instance, it produces more overhead than all other methods combined. Calling this method is unavoidable since it is used to retrieve keys in the MOPMap implementation. However, we need to call this method as little as possible.

While many monitoring components show significant overhead, it is notable that the original program components are also slower when monitoring is present (i.e. 100%). To understand this situation, we analyze the memory usage when monitoring, using Java Management Extensions (JMX) [1]. Table 3 summarizes the memory usage analysis. Monitoring triggers huge memory overheads, resulting in significantly more garbage collection time. With respect to the original program execution time (100%), in monitoring bloat, young object garbage collection takes 278% and full garbage collection takes 258%. In total, garbage collection takes 536% when monitoring bloat and 353% when monitoring pmd. This explains why the original components of the code run far slower when monitoring is present.

We must conclude that the main remaining bottleneck to runtime performance in monitoring is excessive memory usage. Huge memory overhead causes more frequent and longer garbage collections, resulting in larger runtime overhead. We should reduce memory overhead to optimize runtime performance. For example, in Table 2, `WeakReference` object initializations show 6% overhead, while there is no other class ranked in the result. This is because there is a very large number of weak references. We need to reduce the number of objects created for monitoring purposes, especially weak references.

4. Optimizations for Scalability

The more specifications that we monitor simultaneously, the more overhead. Our goal is to improve the overhead in the presence of multiple specifications by finding structures and parts of the monitoring algorithm that may be shared between different specifications. If no specifications overlap with others, in terms of declared events or parameters types, there is nothing much we can improve. Theoretically, the overhead in this case will be the sum of overheads from monitoring them individually. When the memory overhead is excessive, it can be worse than the sum because of the garbage collection behavior.

However, in practice, there are generally multiple specifications for each class, often sharing some events. Among 137 specifications from [26], only 42 specifications are totally independent from all other specifications. Another 95 specifications share parameters or events with some of other specifications. By sharing resources between overlapping specifications we can achieve a truly scalable parametric runtime monitoring system.

In this Section, we explain new techniques for increasing runtime and memory performance first, then we focus on the big picture of the new monitoring mechanism, first presented here, in comparison with the previous monitoring mechanism (Figure 2). Our techniques are formalism-independent and general so that they can be applied to other parametric monitoring systems that use similar indexing tree structures.

4.1 Global WeakReference Table

As explained in Section 2, `WeakReference` is a reference class that refers to an object without blocking it from garbage collection. The indexing tree uses weak references to store parameter objects in its mappings, without blocking garbage collections. In previous versions of JavaMOP, there was no collaboration between specifications, so each specification created a weak reference object for each parameter object. Thus, multiple weak references were potentially created for the same parameter object, if it appeared in different specifications. There is no need to have multiple copies of `WeakReference`; it simply wastes memory.

We introduce a global `WeakReference` table, implemented in the class `GlobalWeakRefTable`, for each parameter type, which all specifications share. This table takes a parameter object as an input and outputs a weak reference. If there is no weak reference in the table for the input object, the table will create one. Thus, weak references will be created only by this table and there will be exactly one copy for one parameter object. Also, upon a non-creation event, we can query the existence of the weak reference without creating one. If there is no weak reference for the parameter object in the table, then there is no monitor in any specification for the parameter object. Thus, we can skip the rest of the steps for checking the existence of monitors for the non-creation event.

The functionality of the `GlobalWeakRefTable` is similar to `HashMap` from the Java API, but its implementation is totally different. If the `GlobalWeakRefTable` stores keys (parameter objects) and values (weak references) in its internal table like `HashMap`, it will cause memory leaks. Instead, the `GlobalWeakRefTable` stores only weak references. Since weak references can refer to the original objects, we can retrieve the weak reference for an object by checking if the weak reference points to the object.

Although the `GlobalWeakRefTable` introduces one more step in the monitoring mechanism, it reduces not only memory overhead by reducing the number of weak references, but also runtime overhead. From the analysis in Section 3, we know that System.identityHashCode() causes the most runtime overhead in the indexing trees. Instead of calling this method in each indexing tree, each `GlobalWeakRefTable` calls this method and stores the result in weak references so that indexing trees can reuse it. To allow this, we implement `MOPWeakReference`, a subclass of `WeakReference` which has a hashcode field, and change the indexing tree to take `MOPWeakReference` as input rather than parameter objects. With this change, indexing trees no longer call the System.identityHashCode() method, removing the main overhead in accessing them. The `GlobalWeakRefTable` calls this method at most once for each parameter object in an event, minimizing the number of the method calls to System.identityHashCode().

The `GlobalWeakRefTable` is essentially the same as the indexing tree except that it does not return monitors. It cleans up references to garbage collected objects and expands the internal data structure just like the indexing tree does [24]. We can reduce overhead even more by combining `GlobalWeakRefTables` with relevant indexing trees, reducing the number of tables and maps. If there is an indexing tree that has the same parameter type at the first level as the `GlobalWeakRefTable`, they can be combined into one data structure. In the majority of cases, `GlobalWeakRefTables` can be combined with indexing trees. Among the many `GlobalWeakRefTables` for the 137 specifications from [26], there are only two `GlobalWeakRefTables` that cannot be combined into indexing trees when monitoring individually, and all `GlobalWeakRefTables` can be combined into indexing trees when monitoring them simultaneously.

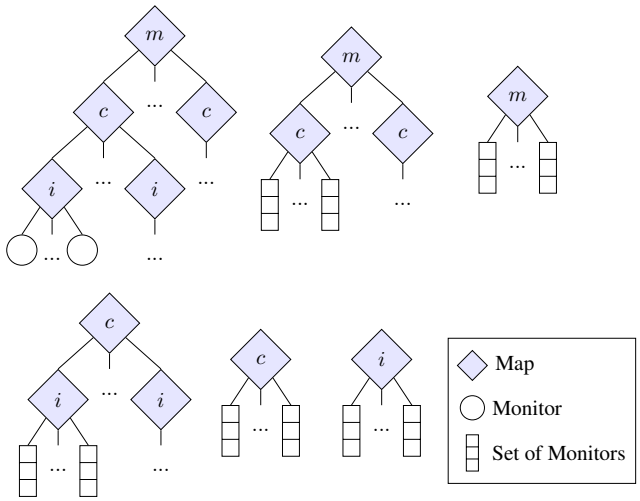

Figure 4. Indexing trees for Map_UnsafeIterator before combining

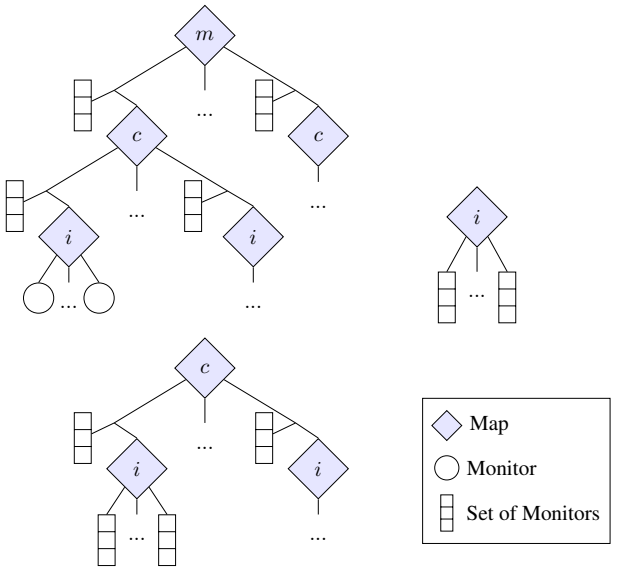

Figure 5. Indexing trees for Map_UnsafeIterator after combining

4.2 Caches for Global WeakReference Table

Under our new technique, the `GlobalWeakRefTable` is the most frequently accessed data structure in monitoring since all events should query this table before accessing any indexing tree. Therefore, it is important to optimize this table. One natural and common method of optimization is caching. In the previous approach, there was already an indexing cache (Figure 2). After adding `GlobalWeakRefTables`, it caches not only a monitor but also weak references for the monitor. Thus, it acts as a cache for both the indexing tree and the `GlobalWeakRefTable`.

Although the indexing cache provides a good cache hit ratio within a specification, it is not good enough when monitoring multiple specifications. First, since there are multiple events from different specifications for the same object, it is likely that multiple specifications consecutively access `GlobalWeakRefTables` for the same object, when their indexing caches miss. Second, the indexing cache is a one-entry cache which is fragile if more than two objects are frequently used together in an interleaved way.

To improve the performance upon this observation, we now use a one-entry level-1 cache to handle the first case and a multi-entry level-2 cache to handle the second case. On a query to the table, we first check the one-entry cache and when it misses, we check the multi-entry cache. However, if we linearly search in the multi-entry cache, the overhead will increase linearly with the number of entries in the cache. Thus, we use a mapping so that we can check only one entry at a time. Because each instrumentation point tends to access the same object consecutively, we index the multi-entry cache by a few least significant bits of the unique id number for instrumentation points, provided by AspectJ. In this way, the multi-entry cache is implemented efficiently. The benefit of the caches surpasses the overhead from maintaining the caches in most cases.

4.3 Combining Indexing Trees

The indexing tree is one of the major bottlenecks in terms of both runtime and memory performance. It contains all of the mappings from parameter objects to monitors. The size of the indexing tree grows as the specification creates more monitors. Additionally, the indexing tree cleans up mappings of garbage collected parameter objects and monitors by itself. Therefore, we can reduce runtime and memory overhead by combining indexing trees. We can combine indexing trees if their defined parameter types share the same prefix. For example, indexing trees for ⟨`Collection`, `Iterator`⟩ and ⟨`Collection`⟩ can be combined but indexing trees for ⟨`Map`, `Collection`, `Iterator`⟩ and ⟨`Collection`, `Iterator`⟩ cannot be combined since the first parameter type, `Map`, appears only in the first.

Combining indexing trees between different specifications is also possible as long as they satisfy the condition for combining. However, it is usually inefficient because there is insufficient mapping overlap between specifications (Section 6). Thus, we combine indexing trees only within each specification. Combining indexing trees in each specification improves not only the performance of monitoring multiple specifications but also the performance of monitoring each specification.

For example, Figure 4 shows all indexing trees for Map_UnsafeIterator before combining them. There are six indexing trees for:

1. ⟨`Map`, `Collection`, `Iterator`⟩
2. ⟨`Map`, `Collection`⟩
3. ⟨`Map`⟩
4. ⟨`Collection`, `Iterator`⟩
5. ⟨`Collection`⟩
6. ⟨`Iterator`⟩

Among six indexing trees, the first three indexing trees can be combined into one, and the fourth and fifth indexing trees can be combined as well. As a result, three indexing trees will remain (Figure 5).

4.4 Specification Activator

In monitoring multiple specifications, such as the 137 specifications from [26], it is common that only some of them are actively monitored when applied to a given program. This is because one program generally does not cover every specification in such a large set of standardized specifications. When a specification does not have any creation event during the execution of a program, it does not need to monitor the program at all. We keep a boolean value as an *activator* for each specification and activate it when there is at least one creation event. When the specification is not activated, we ignore all non-creation events, suppressing the unnecessary overhead. If there is no creation event at all during the execution, all non-creation events will be ignored.

This simple technique successfully deactivates unnecessary specifications during the execution of a program, reducing unnecessary runtime overhead. Even in monitoring a single specification, it can effectively remove unnecessary overhead. In our evaluation (Section 5), some specifications are effectively deactivated and show no overhead at all. The overhead of maintaining specification activators is essentially unnoticeable, far less than the error range of our evaluation (up to 3%).

4.5 Summary of New Monitoring Techniques

Figure 6 summarizes the scalable parametric monitoring mechanism using techniques introduced in this section. Compared to the previous monitoring mechanism (Figure2), there is an activator at the beginning and the `GlobalWeakRefTable` before the indexing tree. Also, instead of parameter objects, it uses weak references in accessing indexing trees.

The main idea of our scalable parametric monitoring is that the `GlobalWeakRefTable` allows sharing of weak references, reducing memory overhead. Also, caching on this table reduces runtime overhead over all specifications using it. Moreover, there are fewer indexing trees and there is no hash method call from the indexing tree. Thus, the overhead from indexing trees has been dramatically decreased. Since the "Copy State" component and the "Create a New Monitor" component also access the indexing tree to add new monitors, overheads from both components decrease as well.

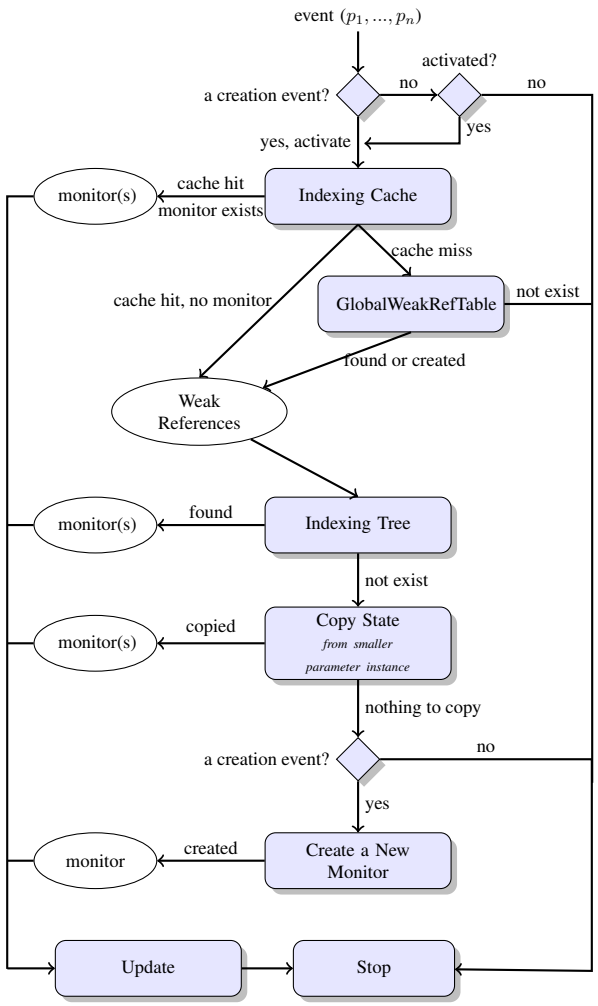

Figure 6. Overview of the scalable parametric monitoring mechanism

5. Evaluation

In this section, we evaluate JavaMOP with the presented scalability improvements on 137 specifications from [26]. Even before this work, JavaMOP had the best runtime performance of any monitoring system, while maintaining competitive memory performance [24]. Also, to the best of our knowledge, there is no other parametric monitoring tool which is capable of practically monitoring 137 specifications simultaneously. Thus, we compare our work on scalability to the previous version of JavaMOP (JavaMOP 2.3).

5.1 Experimental Settings

For our evaluation, we used a Pentium 4 2.66GHz / 2GB RAM / Ubuntu 9.10 machine and Sun JVM 1.6.0_10. For instrumenting benchmark programs with JavaMOP monitoring code, we used version 1.6.11 of the AspectJ compiler (ajc). We monitor 137 specifications for version 9.12 of the DaCapo (DaCapo 9.12) benchmark suite. We also present the result from the `bloat` benchmark in the old version of the DaCapo (DaCapo 2006-10) benchmark suite,

		java.io				java.lang				java.util				All			
# of specs		30				49				58				137			
		Previous		Scalable		Previous		Scalable		Previous		Scalable		Previous		Scalable	
	Orig (sec)	Sum	Together	Sum	Together	Sum	Together	Sum	Together	Sum	Together	Sum	Together	Sum	Together	Sum	Together
bloat	14.4	6	-2	-3	-3	289	327	300	339	1203	1493	719	632	1498	1950	1016	1033
avrora	13.9	7	8	0	5	19	10	12	9	468	336	273	215	494	364	285	235
batik	3.5	0	4	0	2	0	3	0	1	50	37	53	26	50	40	53	26
eclipse	79.5	0	-1	0	2	0	1	42	1	0	1	7	0	0	-2	49	2
fop	2.0	8	7	18	7	96	56	52	55	584	450	380	380	688	N/A	450	N/A
h2	18.7	0	0	13	2	17	24	34	19	71	54	86	47	88	73	133	62
jython	13.6	10	-1	0	1	27	21	18	18	112	90	79	75	149	121	97	95
luindex	2.9	9	5	5	5	5	5	12	3	3	5	5	4	17	9	22	12
lusearch	25.3	14	13	17	16	28	34	21	32	29	26	30	30	71	75	68	79
pmd	8.4	0	-1	0	3	-3	8	0	4	858	898	657	476	855	988	657	486
sunflow	32.5	0	1	0	0	0	1	0	1	4	8	13	6	4	10	13	7
tomcat	14.1	0	-1	0	0	0	0	0	0	0	0	0	-1	0	0	0	1
tradebeans	45.7	40	12	11	2	33	3	26	1	51	1	50	3	124	-1	87	2
tradesoap	95.0	0	2	11	0	16	2	9	1	12	0	4	0	28	0	24	0
xalan	20.9	6	12	-7	30	-17	-12	-32	-16	38	52	31	61	27	34	-8	24

Table 4. Average *percent* runtime overhead for Previous JavaMOP (JavaMOP 2.3) and Scalable JavaMOP (JavaMOP 3.0) (convergence within 3%, N/A: instrumentation crashes)

because it generates large overheads and it is missing in the new version. We used the default data input size, and the -converge option so that the execution time result converges within 3%. AspectJ instrumentation can cause the code to run differently, sometimes resulting in negative overheads even without monitoring. Also, monitoring affects the garbage collection behavior with more memory pressure, often improving garbage collection time; this also accounts for the negative overheads.

All 137 specifications from [26] are based on the Java 6 API documentation concerning three main packages: 30 specifications for `java.io`, 49 specifications for `java.lang`, and 58 specifications for `java.util`. Some specifications are related to the end of the program execution. However, two versions of DaCapo iterate a benchmark program in one execution until the execution time converges. Therefore, we modified those specifications slightly so that they catch the end of iteration of a benchmark program.

5.2 Results and Discussions

Table 4 and 5 summarize the results of the evaluation on the two versions of JavaMOP. Monitoring 137 specifications simultaneously is a considerably challenging task. While monitoring 137 specifications with bloat, there are 839,575,093 events and 27,826,935 monitors created. With pmd, there are 68,438,904 events and 9,510,880 monitors created. Also, in JavaMOP 2.3, 129 indexing trees are required, but the indexing tree combination technique (Section 4.3) reduces the number of indexing trees to 105. Therefore, it is not surprising to see a huge overhead. Although the previous version of JavaMOP was the most efficient parametric monitoring system until this paper, it shows more than 100% overhead on five benchmarks out of 15, including fop.

For fop, the instrumentation crashes because the added instrumentation results in a method larger than the 64KB limit for Java methods. The method size was already too big before the instrumentation, and our instrumentation makes it exceed the limit. In regular programming, the limit of 64KB seems reasonable; any method over 64KB should be re-designed and divided into several methods. However, for procedurally generated code, this limit imposed by Java seems too harsh. While we were unable to obtain overhead for fop with 137 simultaneous specifications in either version of JavaMOP, we do have numbers for monitoring the specification of each package separately.

Table 4 shows the average percent runtime overhead of the two versions of JavaMOP. It shows the sum of overheads for monitoring each specification individually, and the overhead of monitoring them simultaneously, for each benchmark. To avoid the error accumulation, we exclude overheads under 3% for the summation. Overall, Scalable JavaMOP shows significantly less runtime overhead than the previous version of JavaMOP. In monitoring multiple specifications, the previous version of JavaMOP shows higher overheads than the sum of overheads in many places. This is because heavy memory pressure from multiple specifications triggers garbage collection more often. However, Scalable JavaMOP shows much less overhead than the sum of overheads in most cases.

The previous JavaMOP shows 1950% overhead when monitoring all 137 specifications for bloat, while the sum of overheads is 1498%. For pmd, it shows 988% overhead when all specifications are monitored, while the sum of overheads is 855%. However, Scalable JavaMOP shows 1033% and 486% overheads for bloat and pmd, respectively, when all specifications are monitored. These overheads are almost half of what the previous version showed. Also, they are less than the sums of overheads in the Scalable JavaMOP, which are 1016% and 657%, respectively. Note that Scalable JavaMOP also improves the runtime performance of monitoring a single specification, resulting in smaller sums.

		java.io				java.lang				java.util				All			
# of specs		30				49				58				137			
		Previous		Scalable		Previous		Scalable		Previous		Scalable		Previous		Scalable	
	Orig	Sum	Together	Sum	Together	Sum	Together	Sum	Together	Sum	Together	Sum	Together	Sum	Together	Sum	Together
bloat	4.9	5.0	5.7	5.0	4.9	559.2	626.5	586.9	626.8	330.2	1011.2	131.7	282.1	884.6	≥1500	714.5	1466.4
avrora	4.7	4.7	4.5	4.7	4.4	10.9	12.3	7.3	12.5	44.8	73.1	29.5	53.9	51.0	737.2	32.1	198.4
batik	77.3	77.3	76.3	77.3	79.2	77.3	75.1	77.3	72.5	99.2	166.2	88.3	101.7	99.2	166.7	88.3	105.6
eclipse	101.0	101.0	100.0	101.0	99.4	101.0	103.2	101.0	109.1	101.0	113.8	101.0	104.8	101.0	108.0	101.0	113.5
fop	23.9	22.9	25.8	22.0	25.9	79.0	73.2	55.8	58.0	341.4	402.6	255.3	185.2	395.5	N/A	285.5	N/A
h2	267.1	267.1	265.3	267.1	260.9	303.5	327.1	308.0	357.7	2307.9	1176.0	1468.6	957.1	2344.3	1343.5	1509.5	1118.4
jython	21.9	22.1	23.0	21.9	22.9	57.0	76.1	69.2	86.5	91.8	191.8	81.1	71.2	127.1	240.2	128.4	141.2
luindex	6.8	5.7	7.9	8.1	7.0	8.1	18.8	7.0	19.7	6.4	8.8	6.7	12.8	6.6	20.8	8.2	20.8
lusearch	4.6	4.4	4.7	4.0	4.6	4.4	7.5	5.1	7.0	4.8	4.5	5.1	4.9	4.4	7.4	5.0	8.3
pmd	22.3	22.3	25.1	22.3	26.3	87.1	38.5	68.9	38.0	430.5	1474.9	405.4	424.0	495.3	1457.4	452.0	450.8
sunflow	4.5	4.5	4.5	4.5	5.0	4.5	6.6	4.5	6.3	4.7	5.5	4.3	4.4	4.7	7.5	4.3	7.0
tomcat	11.7	11.7	11.7	11.7	12.3	11.7	11.8	11.7	12.3	11.7	11.4	11.7	11.7	11.7	11.9	11.7	11.5
tradebeans	62.9	64.3	63.3	67.6	63.2	66.6	63.1	65.8	63.1	66.3	63.0	64.4	63.9	71.4	63.1	72.0	62.9
tradesoap	63.9	63.9	64.2	63.9	64.1	69.6	62.1	66.0	64.4	68.3	65.4	66.0	65.4	74.0	64.7	68.1	61.4
xalan	4.9	4.9	5.0	5.0	4.9	20.4	21.4	21.3	21.9	4.9	5.0	4.6	4.9	20.4	22.9	21.1	23.7

Table 5. Peak memory usage (in MB) for Previous JavaMOP (JavaMOP 2.3) and Scalable JavaMOP (JavaMOP 3.0) (during 5 iterations, N/A: instrumentation crashes)

Table 5 summarizes the peak memory usage during 5 iterations. In a similar way to the runtime result, it shows the sum of memory overheads from monitoring each specification individually. In the sum of the peak memory usage, the original peak memory usage is counted only once. For example, on bloat, which shows 4.9MB peak memory usage, if two specifications show 5.5MB and 6.2MB peak memory usage, respectively, the sum of peak memory usage is 6.8MB. Overall, the Scalable JavaMOP shows significantly less memory overhead than the previous version of JavaMOP. Similar to runtime performance, Scalable JavaMOP uses less memory, not only when monitoring multiple specifications simultaneously, but also when monitoring them individually. In monitoring specifications individually, in total, Scalable JavaMOP uses 25% less memory than the previous version. In monitoring multiple specifications simultaneously, for avrora and pmd, Scalable JavaMOP shows 3.7 times and 3.2 times less peak memory usage than the previous version, respectively. Also, in total, Scalable JavaMOP uses 34% less memory than the previous version, in monitoring multiple specifications simultaneously.

Monitoring a large number of specifications shows different memory usage from monitoring a single specification. During monitoring process, a large number of objects is generated for the purposes of monitoring. Many of these monitoring objects must be garbage collected. Since the JVM controls the garbage collection throughput so that it does not overwhelm the entire execution time, the garbage collection might not be able to clean up all garbage objects on time. This can cause parameter objects to live longer than usual, delaying accompanied monitoring resources from being garbage collected. In this case, the JVM simply consumes more memory as long as there is more space left. After reaching the memory limit, it starts spending more time for garbage collection. This explains for avrora and others, why monitoring multiple specifications simultaneously shows more peak memory usage than the sum of peak memory usages of individual monitoring and the sum of peak memory usages of monitoring specifications in each package. For example, for avrora and the Scalable JavaMOP, monitoring all specifications in `java.io`, `java.lang`, and `java.util` shows 4.4MB, 12.5MB, and 53.9MB, but monitoring all of the specifications simultatenously shows 198.4MB memory usage at peak.

6. Ineffectual Approaches

In this section, we discuss some ineffectual approaches that we have tried while improving the scalability of parametric monitoring. Although they turn out to be ineffectual in parametric monitoring, some of them might be useful in different settings or they might inspire new effectual ideas.

Combining Indexing Trees between Specifications As mentioned in Section 4.3, we combine indexing trees only within each specification. If we combine indexing trees for different specifications, as well, we can reduce the number of indexing trees even more. However, there is a lot of wasted space in the combined indexing tree. For example, an indexing tree A maps p_1 to m_1 and p_2 to m_2, and another indexing tree B maps p_2 to m_3 and p_3 to m_4. The combined indexing tree of A and B will map p_1 to $(m_1, \varnothing)$, p_2 to (m_2, m_3), and p_3 to $(\varnothing, m_4)$. All empty spaces indicated by $\varnothing$ will be wasted while the indexing trees A and B do not have empty space. More memory overhead from wasted space triggers more garbage collection, slowing down the monitoring.

Enhanced Indexing Cache The indexing cache provides faster retrieval of monitors from the indexing tree. There are several ideas to improve its hit ratio. We can apply a multi-entry cache from Section 4.2. Also, we can cache not only monitors but also lack thereof to save searching the indexing tree for nothing. However, since the index-

ing cache provides already a high hit ratio and the cost to access the indexing tree is already decreased by the `GlobalWeakRefTable`, these enhancements to the indexing cache do not improve the performance. Certainly those ideas increase cache hit ratio, but their benefits are cancelled out by the overheads necessary to support them.

Indexing Tree Cleaning by GlobalWeakRefTables Since we can manage all weak references for each parameter type in one place, the `GlobalWeakRefTable`, we can let the `GlobalWeakRefTable` clean up the indexing trees. In this way, we can remove garbage collected parameter objects from all indexing trees at once, eliminating the need for partial cleanups. Note that partial cleanups could occur even when there is no garbage collected parameter object. We can also have a bit map in the weak reference to indicate to which indexing trees the referent belongs so that we need check only the indexing trees that actually contain it. However, this approach only moves cleanup costs from indexing trees to the `GlobalWeakRefTable`, showing no improvement. The cleanup by the `GlobalWeakRefTable` is more effective because it knows which weak references should be removed. However, cleaning up from outside of the indexing tree costs more because we must locate the entry before we can remove it.

Statistics-Based Indexing Tree Cleaning As mentioned previously, partial cleanups at indexing trees can occur even when there are no garbage collected parameter objects. Since we have the `GlobalWeakRefTable`, we can keep statistics about garbage collected parameter objects and use it for deciding whether to trigger a partial cleanup. However, in most cases, there are garbage collected parameter objects. Saving a relatively small number of partial cleanups does not compensate the overhead necessary.

Event Activator Similar to the specification activator (Section 4.4), non-creation events can be skipped if there is no monitor created for the parameter of the event. However, this approach does not improve the performance because the specification activator already works effectively and the `GlobalWeakRefTable` already returns no weak reference if there was no creation event for the parameter object. Thus, this approach only introduces an overhead of maintaining activators (boolean variables), although the overhead is too small to be notable.

7. Conclusion

Parametric monitoring is a technique for improving the reliability of software that has received an ever increasing amount of attention. Previous work on parametric monitoring has focused on the performance of monitoring single properties in isolation. Realistic uses of monitoring, however, involve monitoring many properties simultaneously, as the large number of properties from [26] can attest. In this paper we have improved the efficiency of JavaMOP with respect to monitoring multiple simultaneous properties; as an added bonus, we also improved performance in the case of a single property. We preformed a thorough analysis of the remaining bottlenecks in the JavaMOP system, and we addressed those that could be addressed without adding more runtime overhead than they save. The cases that were ineffectual, presented in this paper, show that sometimes it is more expensive to address an inefficiency than to let it be. The remaining cases produced real, tangible performance enhancements, in some cases halving overhead in a system that was already heavily optimized.

Acknowledgments

This work is supported by NSF grant CCF-0916893.

References

[1] Java Management Extensions. `http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`.

[2] Hprof: A heap/cpu profiling tool in j2se 5.0. `http://java.sun.com/developer/technicalArticles/Programming/HPROF.html`.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

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

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

[10] 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.

[11] 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.

[12] S. Chaudhuri and R. Alur. Instumenting C programs with nested word monitors. In *Model Checking Software (SPIN'07)*, volume 4595 of *LNCS*, pages 279–283. Springer, 2007.

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

[14] 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.

[15] 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.

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

[17] D. Drusinsky. The Temporal Rover and the ATG Rover. In *Model Checking and Software Verification (SPIN'00)*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.

[18] 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.

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

[20] T. A. S. Foundation. The Apache Commons Collections. http://commons.apache.org/collections/.

[21] 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.

[22] 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.

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

[24] D. Jin, P. O. Meredith, D. Griffith, and G. Roşu. Garbage collection for monitoring parametric properties. In *Programming Language Design and Implementation (PLDI'11)*, pages 415–424. ACM, 2011.

[25] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *J. Formal Methods in System Design*, 24(2):129–155, 2004.

[26] C. Lee, D. Jin, P. O. Meredith, and G. Roşu. Towards categorizing and formalizing the JDK API. Technical Report http://hdl.handle.net/2142/30006, Department of Computer Science, University of Illinois at Urbana-Champaign, March 2012.

[27] 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.

[28] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer*, 2011.

[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.