RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties

Qingzhou Luo[1], Yi Zhang[1], Choonghwan Lee[1], Dongyun Jin[2], Patrick O'Neil Meredith[1], Traian Florin Şerbănuţă[3], and Grigore Roşu[1]

[1] University of Illinois at Urbana-Champaign, Urbana IL, USA
[2] Samsung Electronics, Suwon, Korea
[3] University of Bucharest, Bucharest, Romania
{qluo2,yzhng173,grosu}@illinois.edu, {linjus.yi, pmeredit}@gmail.com,
dongyun.jin@samsung.com, traian.serbanuta@fmi.unibuc.ro

Abstract. Runtime verification can effectively increase the reliability of software systems. In recent years, parametric runtime verification has gained a lot of traction, with several systems proposed. However, lack of real specifications and prohibitive runtime overhead when checking numerous properties simultaneously prevent developers or users from using runtime verification. This paper reports on more than 150 formal specifications manually derived from the Java API documentation of commonly used packages, as well as a series of novel techniques which resulted in a new runtime verification system, RV-Monitor. Experiments show that these specifications are useful for finding bugs and bad software practice, and RV-Monitor is capable of monitoring all our specifications simultaneously, and runs substantially faster than other state-of-the-art runtime verification systems.

1 Introduction

Runtime verification (RV) can increase the reliability of software systems by dynamically verifying property specifications during program execution. Runtime verification has gained significant interest from the research community and there are increasingly broad uses of Runtime Verification (RV) in software development and analysis, as reflected, for example, by abundant approaches proposed recently ([14, 10, 4, 19, 1, 6, 7, 13, 8, 20] among others).

Runtime verification systems typically take a possibly unsafe system as input together with specifications of events and desired properties, and yield as output a modified version of the input system which checks the input specification during its execution, possibly triggering recovery code if specifications are violated. Fig. 1 illustrates this by means of an example using the state-of-the-art monitoring system JavaMOP [20, 15]. An *event* is an abstraction of an action of interest (method invocation, field access, etc.) that occurs during program execution. A *trace* is a finite sequence of events, and a *property* (specification) is a formal description of a set of (desired or undesired) traces. The specification in Fig. 1 has three parameters (line 1): *m*, *c* and *i*. These are bound to concrete objects provided by events at runtime. For example, the event `getI` defined on lines 4–5 is fired when the `iterator()` of an `Iterable` implementation is invoked, and carries two concrete objects that *c* and *i* are bound to. The property, given as an extended

```
1  Map_UnsafeIterator(Map m, Collection c, Iterator i) {
2    creation event getC after(Map m) returning(Collection c) :
3      ( call(Set Map+.keySet()) || ... ) && target(m) {}
4    event getI after(Collection c) returning(Iterator i) :
5      call(Iterator Iterable+.iterator()) && target(c) {}
6    event modifyM before(Map m) :
7      ( call(* Map+.clear*(..)) || ... ) && target(m) {}
8    event modifyC before(Collection c) :
9      ( call(* Collection+.clear(..)) || ... ) && target(c) {}
10   event useI before(Iterator i) :
11     ( call(* Iterator.hasNext(..)) ||
12       call(* Iterator.next(..)) ) && target(i) {}
13
14   ere : getC (modifyM | modifyC)* getI useI*
15               (modifyM | modifyC)+ useI
16
17   @match { print("Map was modified while being iterated"); }
18 }
```

Fig. 1. A JavaMOP specification Map_UnsafeIterator.

regular expression (ERE) (lines 14–15), specifies bad behaviors. The `@match` *handler* (line 17) says what to do when a trace matches the pattern, i.e., when a map is modified while being iterated. Handlers can contain any code, for example, error recovery code.

Despite the usefulness of runtime verification, we believe there are a few hurdles that make developers or users reluctant to use RV systems in practice:

1. It is not easy to write specifications for RV systems to check against;
2. Even if the specifications are available, existing RV systems usually incur large overhead when checking multiple properties simultaneously in real world software.

We next introduce background knowledge and elaborate on the points above in detail.

Lack of specifications: specifications are key to runtime verification. However it is not easy to produce correct specifications, as it requires a deep understanding of the underlying system. There are several hand-written specifications shipped with monitoring systems, as well as a few specifications produced by automated specification mining approaches. These face a few common problems: 1) Correctness. Most specifications are written in ad-hoc ways without rigorous procedures, which make their correctness questionable; 2) Coverage. Most existing specifications only focus on limited coverage of the software system; 3) Reusability. Most existing specifications are specific to the underlying software system. They cannot be applied to different software systems.

High monitoring overhead: producing a highly reliable system while maintaining a low overhead was always a major concern of runtime verification. Most state-of-the-art monitoring systems employ *parametricity* to ensure correctness of the monitored behaviors. Conceptually, parametric monitoring systems maintain a separate trace for each parameter binding and separately check if each trace matches the pattern. As an example in Fig. 1, a `useI` event with an `Iterator` object $i1$ does not affect traces that correspond to the parameter binding where i is bound to $i2$ while m and c can be bound to any object, denoted as $\langle m \mapsto *, c \mapsto *, i \mapsto i2 \rangle$. Instead of maintaining traces, an efficient monitoring system, such as JavaMOP, actually uses a monitor instance for each parameter binding; when an event occurs, it is dispatched to every monitor instance

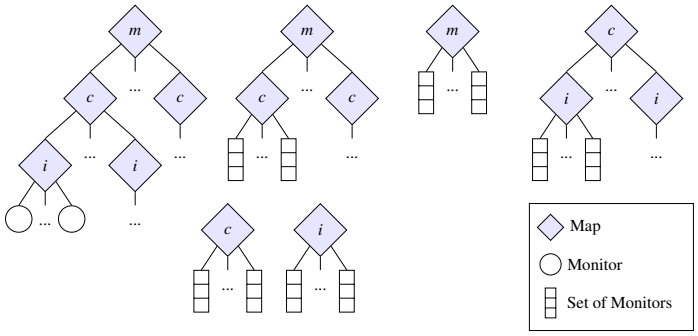

Fig. 2. Indexing trees for Map_UnsafeIterator.

whose parameter binding contains concrete objects carried by the event. If no such instances exist and the event is a *creation event*, a new monitor instance is created.

For fast lookup of monitor instances, JavaMOP employs *indexing trees*, which precisely return all the related monitor instances. An indexing tree is a multi-level map that, at each level, indexes each concrete object of the parameter binding. For example, when a parameter binding is $\langle m \mapsto m1, c \mapsto c1, i \mapsto i1 \rangle$ (m is bound to $m1$, c is bound to $c1$ and i is bound to $i1$), one can retrieve the related monitor instance by searching for $m1$, $c1$ and $i1$ at each level in the 3-level map, shown in the top-left of Fig. 2. However, not all parameters are always bound; e.g., `getI` in Fig. 1 does not carry m, which makes it ineffective to retrieve all the related instances using this map. To handle such case efficiently, another map, shown in the top-right of Fig. 2, is constructed as well. Unlike the 3-level map, where a leaf corresponds to one monitor instance, this 2-level map has a set of instances at each leaf, because there can be multiple parameter bindings that bind c and i to the same `Collection` and `Iterator`.

If an indexing tree holds a strong reference (i.e., an ordinary Java reference) to a concrete object, this object becomes ineligible for garbage collection, which leads to a memory leak. To avoid this, indexing trees store only weak references, which enables the garbage collector to reclaim the referents. Since a weak reference gives indication that the referent has been reclaimed by returning *null*, a monitoring system can detect broken mappings in the indexing trees and clean them up. In addition to indexing trees, an indexing cache that stores the previously accessed monitor instance(s) is used [20].

Even with the use of indexing trees and weak references, the runtime overhead is still large when monitoring multiple parametric properties simultaneously. From a thorough profiling of JavaMOP, [16] identifies the main remaining bottleneck to runtime performance is the excessive memory usage, which is caused by the large size/number of indexing trees. Indeed, when there are hundreds of properties being monitored and each property has a few parameters, the total memory and runtime overhead associated with all the indexing trees will dominate the program execution time. This urges us to design new techniques to further reduce the number and memory overhead of indexing trees.

In this paper, we aim to address the above problems in order to make runtime verification practical. This paper makes the following specific contributions:

- **Comprehensive formal Java API specifications:** We presented a comprehensive set (179 in total) of formal specifications which cover four of the most widely used packages of the Java API. These specifications were manually produced following rigorous procedures, and are publicly available [3].
- **Novel optimization techniques:** We re-implemented JavaMOP, separating its monitoring (as a new RV system, RV-Monitor) from its event firing (as an AspectJ front-end of RV-Monitor). We employ a series of techniques to make RV-Monitor more efficient, especially when monitoring multiple properties simultaneously. Our techniques can also be easily adopted by other RV systems, such as MOPBox.
- **Large scale evaluation:** We monitored all the Java API specifications we produced with RV-Monitor against the DaCapo [5] benchmark suite. Results show that RV-Monitor is capable of finding property violations (potential bugs) in DaCapo while monitoring hundreds of properties at the same time. Our comparison also shows that RV-Monitor runs substantially faster than other monitoring systems.

2 Formal Specifications from the Java API Specification

Almost all Java programs make use of the Java API. Moreover, misuses of the Java API can result in runtime errors or nondeterministic behavior. It is therefore important to obey the usage protocols of the Java API. In this section we describe our formalization of several Java API specifications. All our tools and specifications are publicly available and can be used and verified by any RV systems [3].

2.1 Java API Specification

A Java platform, such as Java Platform Standard Edition 6, implements various libraries that are commonly needed to implement applications, such as data structures (e.g., `List` and `HashMap`), and I/O functions (e.g., `FileInputStream` and `FileOutputStream`). Besides such library implementations, a Java platform provides the *API Specification*, which describes all aspects of the behavior of each method on which user's programs may rely. For example, the API specification for the `Map` interface states:

> If the map is modified while an iteration over the set is in progress ... the results of the iteration are undefined.

We believe the Java API documentation is a good source for formal specifications. First of all, the Java API is commonly used by virtually all Java programs. Thus the use protocols of those API methods should always be obeyed. Second, the Java API specification is well written, clearly stating what the correct/incorrect usage of API methods is. For example, in the above `Map` specification it can be inferred that any methods that modify the `Map` (`put(), remove(), etc.`) should *not* be called *before* the iteration of the `Map` is done. Violations of such properties usually indicate a bug or bad programming practice.

We have carefully read and analyzed the complete Java API documentation for four of the most commonly used packages: `java.io`, `java.lang`, `java.net` and `java.util`. We employed a multi-step approach to formalizing all the specifications in these four packages and produced 179 formal specifications. Next we describe our approach.

2.2 Separating Specification-implying Text

Although the API specification includes contracts (Sec. 2.1), sentences referring to different purposes are mingled in API specifications. As a preparation step, we marked specification-implying text versus descriptive text using a specially defined javadoc tag. This facilitates the review of relevant parts when writing specifications (Sec. 2.3).

While it might seem obvious to distinguish between descriptive and specification-implying text, there are unclear cases. One case is the description of conditions involving external environments. Here is an example for the `FileOutputStream` constructor:

> If the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason, then a `FileNotFoundException` is thrown.

We decided to not formalize this property, because the state of the file system can externally and dynamically change without notifying the RV system and, consequently, it is impossible to reliably check whether a file can be created or opened.

It is difficult to formalize a specific set of rules that resolves all of the unclear cases, but the rule of thumb was that a chunk of text is specification-implying only if a desirable or undesirable behavior is apparent and it is defined in terms of noticeable events, such as class loadings, method invocations and field accesses.

2.3 Writing Formal Specifications

For each chunk of text that we marked as specification-implying, we wrote JavaMOP specifications. We chose the JavaMOP specification syntax because JavaMOP was the most efficient and expressive, in that it allows us to write a property in various formalisms. As shown in Fig. 1, a typical specification contains three parts: a set of event definitions, a desirable or undesirable behavioral pattern (i.e., property), and a handler for violations. An event in our specifications is mostly a method invocation, a field access, an end of an execution, or a construction of an object. Depending on the pattern, we chose the most intuitive formalism, among an ERE, a finte-state machine (FSM) or a linear temporal logic (LTL) formula, for expressing a property. Our handlers simply output a warning message in case of a violation.

There are some cases where we intentionally did not write formal specifications. Below we explain such cases with rationales.

Non-monitorable behaviors We considered only runtime-monitorable specifications because we intended to use a RV system; e.g., consider the following:

> The implementor must also ensure that the relation is transitive: `(x.compareTo(y) > 0 && y.compareTo(z) > 0)` implies `x.compareTo(z) > 0`.

Although this implies a certain behavior, checking whether it holds is infeasible at runtime. Not having a means of describing and checking it, we did not write formal specifications for such cases.

Already enforced behaviors We did not write formal specifications if the desirable behavior is already enforced by compilers. For example, consider the following, which states the requirement of `InputStream`'s subclasses:

> Applications that need to define a subclass of `InputStream` must always provide a method that returns the next byte of input.

Java compilers enforce the requirement because `read()`, the method implied by the comment, is an abstract method. Such guarantee obviates the need for additional runtime specifications; thus, we did not formalize such cases.

Internal behaviors When all the events in the implied specification are never exposed to clients (e.g., they are private method invocations or field accesses), we did not write specifications. For example, consider the API specification for `GregorianCalendar.getYearOffsetInMillis()`:

> This `Calendar` object must have been normalized.

A specification on this is feasible but useless because user's programs cannot invoke it anyway, due to the access control—it is defined as `private`. We did not write specifications in such cases because there is no benefit from a user's perspective.

2.4 Classifying Formal Specifications

Not all violations are equally important; violations of some specifications are harmless in certain applications and users may want to suppress them. To facilitate such filtering, we classified our specifications into three groups: suggestion, warning and error. We use suggestion if a violation is merely a bad practice. If a violation is not necessarily erroneous but potentially wrong, we use warning. We use the last group, error, if a violation indicates an error.

3 Scalable Runtime Verification

Many RV systems are able to monitor *one* property efficiently. However, to make runtime verification practical, an RV system should be able to efficiently monitor *multiple* properties simultaneously. In this section we present our optimizations to reduce runtime overhead. Since JavaMOP was the most efficient RV system when this paper was written, we built upon several of its optimizations, especially upon its indexing tree idea. Most of our techniques focus on making the indexing tree accesses faster, which reduce the monitoring overhead when multiple properties and monitor instances are present.

Note that the idea of using indexing trees and caches for managing multiple monitor instances is commonly used among other RV systems. For example, MOPBox uses JavaMOP's indexing algorithm to map variable bindings to monitors. We believe our improvement of JavaMOP, such as the merging of indexing trees, can also be directly applied to other RV systems and can indirectly inspire other systems, such as the more general RuleR or Eagle [4, 9]. Besides, some other systems propose static analysis on top of the JavaMOP technique; those systems will also benefit from our techniques.

3.1 Global Weak Reference Table

Even monitoring a single specification causes multiple weak references for a single object in JavaMOP because it constructs multiple indexing trees for a specification and each indexing tree stores its own copy of the weak reference. From the specification shown in Fig. 1, for example, JavaMOP can create four weak references for a single `Collection` object: one at the second level of the top-left tree, another at the first level of the bottom-left tree, and so forth in Fig. 2.

When multiple specifications are monitored, redundancy is more severe because JavaMOP creates separate indexing trees for each specification. That is, indexing trees for another specification create their own (possibly multiple) weak references for the same object, in addition to those already created for the first specification. For example, both Map_UnsafeIterator and Iterator_HasNext—it states that `Iterator.next()` should not be called without checking for the existence of the next element—create their own weak references to the same `Iterator` objects.

We avoid such intra/inter-specification redundancy by employing a global weak reference table (GWRT) that stores only one weak reference for each distinct object. As in string interning, however, keeping only one copy requires extra computation for creation—it is necessary to check whether or not a weak reference for the given object has been previously created, in order to guarantee uniqueness of weak reference. To eliminate redundancy with little performance compromise, we use a separate GWRT for each parameter type; e.g., three GWRTs will be used by Map_UnsafeIterator. A GWRT functions as a dictionary where a concrete object is used as a key and a weak reference is used as a value. That is, this table allows one to retrieve the weak reference associated with the concrete object. Since this table is globally shared among all specifications and a key can appear at most once in it as in other typical dictionaries, it allows only a single copy of weak reference for each object.

While the GWRT may seem to be the same as any other hashtable with weak reference values, its internal structure is further optimized to reduce memory consumption. First, it does not hold a strong reference to a concrete object, which would cause memory leaks. More importantly, this table does not actually store pairs of objects and their weak references; only weak references are stored because one can retrieve the object from the weak reference by invoking `WeakReference.get()`. Fast lookup, achieved using a hash function, is still viable by locating the slot based on the hash value of the object and searching weak references in the slot.

We further reduce the memory overhead incurred from GWRTs by merging them with the first levels of indexing trees. The first level of an indexing tree holds all the weak references to the objects of a certain type, as does the GWRT for that type. Therefore, the GWRT can be embedded in the first level of the indexing tree, without losing any weak reference, introducing wasted space, or compromising fast lookup. The GWRT for `Map`, for example, is embedded in the first level of the top-left tree in Fig. 2, which holds entries for all the weak references, one per each `Map` object. Embedding eliminates all memory overhead of said GWRT. Most GWRTs can be embedded because RV-Monitor needs to create indexing trees with many different types. For example, RV-Monitor creates at least one indexing tree for each parameter type for Map_UnsafeIterator, as shown in Fig. 2, which allows the three GWRTs to be embedded. In our experiments, all GWRTs

were embedded when the 137 specifications from [18] were monitored simultaneously. However, there are some theoretical cases where embedding is not possible.

Moreover, reducing the number of invocations of `System.identityHashCode()` is also beneficial. Therefore, a subclass of the `WeakReference` class, Java's weak reference implementation, was created in such a way that the hash value of the referent is computed at most once and then stored in a dedicated field by the constructor. This subclass is used in place of the `WeakReference` class in both GWRTs and indexing trees. With this subclass, calls to `System.identityHashCode()` can be replaced by a field access, which is less expensive, at the cost of adding one `int` field to each weak reference. Adding this field to each weak reference does not cause a significantly larger expenditure of memory because only a single weak reference is created for each distinct object thanks to GWRTs.

Another benefit of GWRTs is that it is easier to check if there exists a monitor instance referring to a given object. Because a GWRT holds all the weak references of one type, the lack of the weak reference for an object implies that none of specifications have created any monitor instance for that object. For this reason, when handling a non-creation event (see Section 1), we can safely skip the other steps of monitoring if there are no weak references for every parameter carried by the event. In contrast, JavaMOP, where indexing trees are separately created for each specification, must check indexing trees for each specification that shares a given event.

3.2 Caches for the Global Weak Reference Table

GWRTs are frequently accessed because weak references for each parameter object bound in a given event must be retrieved. As a result, optimizing this data structure has a large impact on performance. We first reduce the number of accesses to a GWRT by extending each indexing cache. Temporal locality is often exhibited in the access patterns of objects related to monitoring a given specification. For example, consider a specification that formalizes a resource management pattern on the `Reader` class: "a reader can perform the read operation until it is closed." While monitoring this specification, it is likely that there would be a long sequence of `read()` calls on a particular `Reader` object until `close()` is called. To utilize such locality, JavaMOP already has an indexing cache that stores the recently accessed monitor instance(s).

While this cache allows for quick retrieval of the monitor instance(s), weak references for all of the parameter objects associated with those instance(s) are often necessary during the process of updating said instance(s). Quick retrieval of a monitor instance followed by retrieving weak references for every single parameter object from the GWRT would be contrary to the whole purpose of the indexing cache. We therefore extended the indexing cache in such a way that, when it caches monitor instance(s), it also caches the weak references that point to the parameter objects bound by those monitor instance(s). As a result, when an event carries the recently handled objects, the indexing cache can return not only the relevant monitor instance(s) but also weak references, without the need to consult the GWRT for each bound parameter object.

Second, we add to each GWRT a cache, called the GWRTCache, which is used to store recently requested weak references. As stated previously, multiple specifications commonly share event definitions, thus a single method invocation can trigger multiple

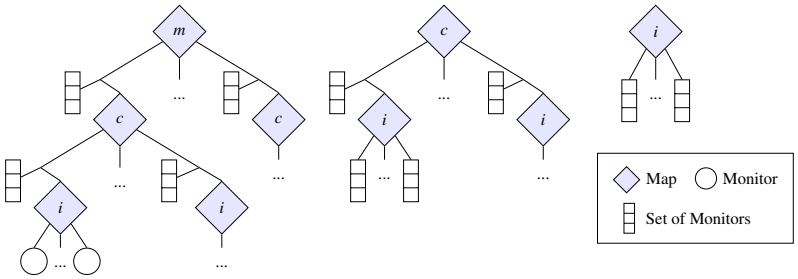

Fig. 3. Indexing trees for Map_UnsafeIterator after combining.

events, one per each specification. For example, an invocation of `Reader.read()` triggers not only an event of the aforementioned resource management specification but also the "mark/reset" specification: "a reader's mark position, set by `Reader.mark()`, is invalidated after reading the specified number of bytes." Without the GWRTCache, when `read()` is invoked on an unprecedented `Reader` object, indexing caches for both specifications will miss—while handling the two events—and both caches would query the GWRT for the `Reader` object, in order to create or retrieve the weak reference (whichever request occurs first will create the weak reference). Using the GWRTCache, however, the first miss causes this cache to hold the created `Reader` object and, consequently, a hash lookup in the GWRT for each subsequent request is replaced by a lookup in this cache, which is faster.

3.3 Combining Indexing Trees

As we mentioned in Section 1, monitoring a real program causes millions of monitor instances and, consequently, the size of indexing trees becomes large because they store references to monitor instances at leaves. In the presence of hundreds of specifications, the size of indexing trees is likely to become even larger, causing excessive memory overhead. To mitigate such overhead, we combine indexing trees originating from the same specification, if they share the same prefix; e.g., indexing trees for ⟨`Collection`, `Iterator`⟩ and ⟨`Collection`⟩ are combined, but trees for ⟨`Map`, `Collection`, `Iterator`⟩ and ⟨`Collection`, `Iterator`⟩ are not.

Consider the indexing trees for ⟨`Collection`, `Iterator`⟩ and ⟨`Collection`⟩, shown in Fig. 2. Since both trees index all the `Collection` objects at the first level, their first level maps contain the exactly same keys (i.e., weak references); the only difference is that, by the same key, a second level map is mapped to in the first tree, whereas a set of monitor instances is mapped to in the second tree. We combined these two trees by allowing a second level map and set of monitor instances pair to be mapped to by a key, as shown at the bottom tree of Fig. 3. Choosing which part of the pair can be done depending on whether or not a `Collection` object is carried by an event. Three indexing trees, shown at the top of Fig. 2, are also combined into one tree, shown at the left top of Fig. 3, similarly. By combining compatible trees, only three trees, out of six, remain as shown in Fig. 3.

3.4 Fine-grained Locking

JavaMOP uses one global lock throughout all the operations for handling an event, which may involve multiple GWRT accesses and indexing tree lookups. This can significantly hinder concurrent execution in the presence of multiple specifications because it is likely that more events occur simultaneously.

To reduce this hindrance, we remove the global lock, and instead use fine-grained locking given that each of the GWRTs and indexing trees is independently accessed. First, we synchronize each GWRT separately because GWRTs do not interfere with each other. This enables multiple threads to run concurrently, unless they handle the same type of parameters—recall that one GWRT is created for each parameter type. Second, we synchronize each level of an indexing tree separately. For example, consider a `getC` event in Fig. 1, which brings two parameters: *m* and *c*. When looking up the left tree in Fig. 3 for retrieving monitor instances corresponding to the provided parameters, we first acquire a lock corresponding to the first level. On retrieving the node at the second level according to the object bound to *m*, we immediately release the lock. This way, another request on the first level of this tree can be served with relatively short delay.

In order to promote concurrent execution further, we moved to thread-local storage (TLS) two caches: the cache in each indexing tree and the GWRTCache in each GWRT (Section 3.2). This is based on the observation that most objects bound to parameters are solely used in a single thread. With this change, if a request is served by the cache, no synchronization is performed, at the cost of adding a few cache entries to each GWRT and each level of indexing trees, per thread.

4 Implementation and Evaluation

In this section we describe our implementation of RV-Monitor and evaluate the correctness of all our formal specifications as well as the performance of RV-Monitor.

4.1 Implementation

We have developed RV-Monitor in such a way that it can be used as a universal platform for building various monitoring systems. To achieve this, we re-implemented JavaMOP and separated event monitoring functionality from event firing functionality. RV-Monitor implements only the common and indispensable monitoring functionality-such as listening to events and triggering handlers when a pattern matches. Using RV-Monitor, one can generate a monitoring library that exposes an interface—more specifically, a set of methods, each of which corresponds to an event definition—for listening to events. One can then monitor a program by notifying the library of events; i.e., inserting invocations of such methods either manually, or systematically using instrumentation tools. Having this separate system enables one to write a specification in terms of virtually any event, from method invocation to monitor (in the context of locking) wait, using diverse means, such as bytecode instrumentation or JVMTI. No matter how events are fired, monitoring can be efficient because it is entirely delegated to the code generated by RV-Monitor.

We also re-implemented the event firing functionality of JavaMOP as an AspectJ front end of RV-Monitor. All the original JavaMOP specifications can be translated into

simple AspectJ code. Inside those AspectJ code, RV-Monitor event monitoring library methods are now called. The AspectJ front end of RV-Monitor enables us to easily compare the performance of RV-Monitor with JavaMOP.

4.2 Correctness of Formal Specifications

The correctness of specifications is vital, because incorrect specifications would miss bugs or give out false alarms. However there is no silver bullet to guarantee the correctness of specifications other than careful inspection and thorough testing. All of our specifications were reviewed by at least two authors who are knowledgeable about Java. In addition to peer review, we wrote small defective Java programs and tested if the formal specification can reveal violations. We have written more than 100 (publicly available) small programs in total, and all the tests revealed the inserted defects, which brings evidence that these specifications are capable of detecting errors. All our specifications and tests can be found at [3].

4.3 Runtime Overhead

Comparison with JavaMOP To measure the efficiency of RV-Monitor, we compare the overhead of RV-Monitor (with the AspectJ front end) with that of the latest version of JavaMOP (v2.3). We chose JavaMOP because, at the time of writing this paper, it was the most efficient system, to the best of our knowledge. All our experiments were conducted on Sun Java SE 6 (build 1.6.0_35) under a system with a 3.5 GHz Intel Core i7 and 32GB of memory running Linux 3.2.

We collected the execution time for each benchmark, as shown in Table 1, under a steady state using DaCapo 9.12's `-converge` option. We took the average time of the last five runs. The "Original" column shows the unmonitored runs of benchmarks, whereas the other columns show the monitored runs with all the specifications. The "overhead" columns indicate the percentage overhead.

In average, the percentage overhead of RV-Monitor is less than half of that of JavaMOP. In particular, improvement was significant for four benchmarks that caused excessive overheads under JavaMOP; avrora, lusearch, pmd and xalan under JavaMOP respectively showed 150%, 654%, 797% and 2,968% overheads, but they caused only 22%, 420%, 261% and 97% overheads under RV-Monitor. On one benchmark, fop, RV-Monitor was noticeably worse. This is because fop is single-threaded, and RV-Monitor has been geared toward increasing concurrency, such as fine-grained locking and thread-local storage (TLS) caches (Section 3.4), which are beneficial to multi-threaded programs but may end up adding extra computation when concurrency is not needed.

Note that in xalan, RV-Monitor performs much better than JavaMOP. The reason is that the program itself is computation intensive. Many monitor instances are created and used only once, incurring a large memory overhead with JavaMOP. Therefore, RV-Monitor is able to greatly reduce the overhead by merging indexing trees and using global caches. If we exclude this program, the average overhead of RV-Monitor will be 143%, still better than JavaMOP with 163%. We believe the DaCapo benchmark has been conceived as a whole, to test various aspects of the language, and established itself as the most suitable benchmark for runtime verification. Therefore, we think the huge overhead reduction case shows a big improvement of our technique, not an isolated case.

Benchmark	Original	JavaMOP†		RV-Monitor		MOPBox		
	time (s)	time (s)	overhead	time (s)	overhead	times (s)	overhead	GCs
avrora	4.55	11.36	150%	5.55	22%	>600	-	142
batik	0.80	0.96	20%	1.27	59%	>600	-	195
eclipse	10.77	10.51	-2%	10.68	-1%	>600	-	216
fop	0.23	0.92	300%	1.85	704%	>600	-	193
h2	4.59	6.44	40%	10.64	132%	>600	-	176
jython	1.33	2.98	124%	3.84	189%	>600	-	209
luindex	0.56	0.60	7%	0.57	2%	383.76	68428%	199
lusearch	0.50	3.77	654%	2.60	420%	>600	-	111
pmd	1.71	15.34	797%	6.17	261%	>600	-	198
sunflow	1.31	1.77	35%	1.57	20%	>600	-	73
tomcat	1.36	1.36	0%	2.09	54%	1.37	0%	18
tradebeans	6.52	6.33	-2%	6.91	6%	6.84	4%	44
tradesoap	3.88	3.91	1%	3.83	-1%	3.70	-4%	87
xalan	0.38	11.66	2,968%	0.75	97%	>600	-	88
Average			364%		140%		-	

Table 1. Execution time and the number of allocations during execution.

It should be noted that the runtime overhead may look high, for some benchmarks, but those cases are extreme: here we monitored programs that intensively use the Java API against lots of specifications, and this is indeed a challenging task. For example, most benchmarks emitted millions of events; in particular, avrora, h2 and pmd emitted 32,804,400, 65,647,663 and 48,866,293 events, respectively. In usual cases, however, one can expect moderate overhead.

Comparison with MOPBox MOPBox [21] is the most similar RV system to RV-Monitor. MOPBox [21] requires one to construct an FSM by setting alphabets, states, and transitions using its API and it does not support other formalisms. Because of that, we tested only the most heavily used specification in DaCapo, Collection_UnsafeIterator, which warns if a collection is modified while an iterator is being used.

Table 1 shows the execution time and the number of garbage collections. Even though only one specification was monitored, most benchmarks barely finished the first iteration in 10 minutes, except three benchmarks that fire relatively few events: tomcat, tradebeans and tradesoap. One of reasons for such excessive overhead was memory consumption: we noticed that an execution under MOPBox triggered garbage collections frequently as shown in Table 1, whereas that under RV-Monitor triggered merely at most 5 garbage collections, for each iteration, in the worst case.

Overall, the result shows that the overhead of MOPBox with one specification is by far more than that of RV-Monitor with 179 specifications. There could be many possible reasons for the big difference. First, RV-Monitor enables the monitored program to directly pass parameters to events (through generated AspectJ code), while MOPBox requires a `VariableBinding` object to be created for each parameter. This may cause extra runtime overhead. Also, MOPBox uses the $\mathbb{C}^+\langle X\rangle$ algorithm [8], which is less

Package	io			lang			net			util		
Severity	err.	warn.	sugg.	err.	warn.	sugg.	err.	warn.	sugg.	err.	warn.	sugg.
# of specs	19	6	5	23	11	14	31	12	1	43	11	3
# of violated specs	3	1	3	2	0	11	1	2	0	6	4	1
# of violations	19	14	12	36	0	4724	3	2	0	14	134	60

Table 2. The number of specifications, violated specifications and violations.

efficient than $\mathbb{D}\langle X\rangle$ [8], used by RV-Monitor. Moreover, MOPBox does not use all the efficient data structures for handling monitor instances, such as indexing trees.

We believe that our comparison with MOPBox, as well as that with JavaMOP, shows that engineering a high-performance runtime verification system for parametric specifications is a highly challenging task.

4.4 Bug Finding

Runtime monitoring is capable of finding bugs and bad practices even from widely used benchmarks. Compared with JavaMOP, RV-Monitor is able to monitor all the properties simultaneously on large programs, makes it practical for finding bugs. Table 2 summarizes the number of specifications, the number of violated specifications, and the number of violations for each severity level from all the benchmarks of DaCapo 9.12. When we counted the number of violations, we counted all violations caused by the same call site as one. Here we focus on the violations of specifications that are marked as error and warning among thousands of violations.

Reader_ManipulateAfterClose, which warns if a read operation is performed after a `Reader` object has been closed, was violated by 13 out of 14 benchmarks of DaCapo. In fact, `read()` failed and the reader was immediately closed, but `read()` was invoked again on that closed reader. The latter `read()` call was reached because the exception raised at the first failure was discarded by a method and it returned as if there were no errors. Since the `Reader` implementation raises `IOException` anyway and it is properly handled, this violation does not result in a notable failure. Nevertheless, we believe it is a bad practice to rely on an exception even when a violation is predictable.

ShutdownHook_LateRegister, which warns if one registers or unregisters a shutdown hook [4] after the Java Virtual Machine (JVM)'s shutdown sequence has begun, was violated by h2—this program attempted to unregister a shutdown hook. One may think that such attempt would be safe as long as the resulting exception is properly handled, but it is unsafe because registered hooks are started in unspecified order and, consequently, the hook to be unregistered may have been already started.

Collections_SynchronizedCollection, which warns if a synchronized (thread-safe) collection is accessed in an unsynchronized manner, was violated by jython. This program created a synchronized collection, using `Collections.synchronizedList()`, but iterated over the collection without synchronizing on it, which may result in nondeterministic behavior, according to the API specification.

Besides these violations that imply notable problems, we also found many minor yet informative violations that static analysis might not be able to detect. One such

[4] According to the API specification, a *shutdown hook* is an initialized but unstarted thread.

example is a violation of Math_ContendedRandom, which recommends one to create a separate pseudorandom-number generator per thread for better performance if multiple threads invoke `Math.random()`. Another example is StringBuffer_SingleThreadUsage, which checks if a `StringBuffer` object is solely used by a single thread. To detect such violations without false positives, it is necessary to accurately count how many threads access an object or a method, which is impossible for static checkers in full generality.

4.5 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 for 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 3.3, 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 3.2. Also, we can cache not only monitors but also lack thereof to save searching the indexing tree for nothing. However, since the indexing cache already has a high hit ratio, these enhancements do not improve the ratio enough to justify their overhead.

Indexing Tree Cleaning by GWRT Since we can manage all weak references for each parameter type in one place, the GWRT, we can let the GWRT 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 GWRT, showing no improvement. The cleanup by the GWRT 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 GWRT, 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.

5 Related Work

Producing Specifications Many automated specification mining approaches [2, 12, 11, 17] has been proposed to reduce the considerable human effort in writing specifications. Unfortunately, they tend to have a few problems in general. First, many of them require or assume non-trivial inputs to yield a meaningful specification [2]. Second, many approaches support only very particular properties [12, 11]. Last but not least, most of them do not guarantee correctness of their results [17].

Monitoring Specifications Many approaches have been proposed for runtime verification [10, 4, 19, 1, 6, 7, 13]. Most of them do not focus on reducing the overhead of monitoring multiple properties simultaneously. Recently Purandare et al. [22] presents the first study of overhead arising during the simultaneous monitoring of multiple finite-state machine (FSM) specifications. Their approach reduces runtime overhead by merging FSM monitors into bigger monitors, resulting in significant runtime overhead reduction (~50%) over JavaMOP v2.1; it is unknown how that compares with the more recent JavaMOP v2.3, which added several optimizations over v2.1 that minimize the number of created monitors. Their monitor-merging idea is orthogonal and complementary to ours techniques of merging indexing trees and using global caches, as our techniques do not take into account the internal structure of each monitor.

Except for that work, most existing monitoring systems are not capable of handling multiple specifications simultaneously, or have rudimentary support, considering each specification separately. Since each specification is individually handled, the runtime overhead for running them simultaneously is likely to be at least the summation of the overheads of running each in isolation. In fact, our preliminary experiment with JavaMOP (v2.3) showed that the overhead of monitoring several specifications simultaneously is higher than the summation of the overheads of monitoring each in isolation, probably because more memory pressure and less cache hit.

6 Conclusion

Runtime verification has not been widely adopted by developers and users, mainly because of lack of usable specifications and large runtime monitoring overhead. In this paper we have presented a set of 179 formal specifications covering most commonly used Java packages. We have also employed a series of techniques to decrease runtime overhead of monitoring multiple properties simultaneously, which resulted in a new monitoring system, RV-Monitor. Results showed that: 1) our specifications help finding bugs or bad practice coding in real world benchmarks; 2) our techniques make RV-Monitor runs significantly faster than other state-of-the-art RV systems.

Acknowledgement: the work presented in this paper was supported in part by the Boeing grant on “Formal Analysis Tools for Cyber Security” 2014-2015, the NSF grants CCF-1218605, CCF-1318191 and CCF-1421575, and the DARPA grant under agreement number FA8750-12-C-0284.

References

[1] 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 *OOPSLA*, 2005.

[2] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, 2002.

[3] Annotated Java API Specifications. `https://code.google.com/p/annotated-java-api/`.

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

[5] 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 *OOPSLA*, 2006.

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

[7] S. Chaudhuri and R. Alur. Instumenting C programs with nested word monitors. In *SPIN*, 2007.

[8] F. Chen, P. Meredith, D. Jin, and G. Rosu. Efficient formalism-independent monitoring of parametric properties. In *ASE*, 2009.

[9] M. d'Amorim and K. Havelund. Event-based runtime verification of Java programs. *SIGSOFT Softw. Eng. Notes*, 2005.

[10] D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN*, 2000.

[11] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE*, 2000.

[12] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *ICSE*, 2008.

[13] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *OOPSLA*, 2005.

[14] K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In *RV*, 2001.

[15] D. Jin, P. O. Meredith, D. Griffith, and G. Roşu. Garbage collection for monitoring parametric properties. In *PLDI*, 2011.

[16] D. Jin, P. O. Meredith, and G. Roşu. Scalable parametric runtime monitoring. Technical Report http://hdl.handle.net/2142/30757, Department of Computer Science, University of Illinois at Urbana-Champaign, 2012.

[17] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *ICSE*, 2011.

[18] 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, 2012.

[19] M. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA*. ACM, 2005.

[20] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *STTT*, 2011.

[21] MOPBox. `https://code.google.com/p/mopbox/`.

[22] R. Purandare, M. B. Dwyer, and S. G. Elbaum. Optimizing monitoring of finite state properties through monitor compaction. In *ISSTA*, 2013.