Can We Monitor All Multithreaded Programs?

Antoine El-Hokayem, Yliès Falcone

Can We Monitor All Multithreaded Programs?

Antoine El-Hokayem and Yliès Falcone

Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France
`firstname.lastname@univ-grenoble-alpes.fr`

Abstract. Runtime Verification (RV) is a lightweight formal method which consists in verifying that an execution of a program is correct wrt a specification. The specification formalizes with properties the expected correct behavior of the system. Programs are instrumented to extract necessary information from the execution and feed it to monitors tasked with checking the properties. From the perspective of a monitor, the system is a black box; the trace is the *only* system information provided. Parallel programs generally introduce an added level of complexity on the program execution due to concurrency. A concurrent execution of a parallel program is best represented as a partial order. A large number of RV approaches generate monitors using formalisms that rely on total order, while more recent approaches utilize formalisms that consider multiple traces.

In this tutorial, we review some of the main RV approaches and tools that handle multithreaded Java programs. We discuss their assumptions, limitations, expressiveness, and suitability when tackling parallel programs such as producer-consumer and readers-writers. By analyzing the interplay between specification formalisms and concurrent executions of programs, we identify *four* questions RV practitioners may ask themselves to classify and determine the situations in which it is sound to use the existing tools and approaches.

1 Introduction

Analyzing and verifying programs typically relies on an abstraction of the program execution. One such abstraction, a *trace*, focuses on parts of the executed program. Traces typically contain operations and events that a program executes. They are versatile: they serve to analyze, verify and characterize the behavior of a program. A single trace records information of a program execution. Information serves to profile the run of a program [1] so as to optimize its performance. Alternatively, a trace abstracts a single program execution, to verify behavioral properties expressed using formal specifications. A collection of traces model the program behavior as it allows to reason about possible executions or states. As such, multiple traces serve to check for concurrency properties [48] such as absence of data races [42,57] and deadlock freedom [39].

Runtime Verification (RV) [31,46,9] is a lightweight formal method which consists in verifying that an execution of a program is correct wrt a specification. The specification formalizes with properties the expected correct behavior of the system. Programs are instrumented to extract necessary information from the execution and feed it to monitors. This information is typically referred to as the *trace* [56]. Monitors are synthesized from behavioral properties, they check if the trace complies with the properties. From the monitor perspective, the system is a black box; the trace is the *sole* system in-

Listing 1.1: A shared queue for *producer-consumer*.

```
1  public class SynchQueue {
2    private LinkedList<Integer> q = new LinkedList<Integer>();
3    public void     produce(Integer v)   { q.add(v); }
4    public Integer consume()       { return q.poll(); }
5  }
```

Thread 0 (Producer)	*Thread 1 (Consumer)*
① sq.produce(0);	② sq.consume(); //0
③ sq.produce(1);	④ sq.consume(); //1

Fig. 1: Operations for a single producer and a single consumer thread operating on a shared queue (`sq`). Shaded circles specify a given number associated with the statement.

formation provided. Therefore, for any RV technique, providing traces with correct and sufficient information is necessary for sound and expressive monitoring[1].

Parallel programs introduce an added level of complexity because of concurrency. The introduction of concurrency can result in the collected trace not being representative of the actual *concurrent execution* of a parallel program. A concurrent execution is best modeled as a partial order over *actions* executed by the program. The actions can represent function calls, or even instructions executed at runtime. The order typically relates actions based on time, it states that some actions happened before other actions. Actions that are incomparable are typically said to be *concurrent*. This model is compatible with various formalisms that define the behavior of concurrent programs such as weak memory consistency models [2,3,49], Mazurkiewicz traces [50,36], parallel series [47], Message Sequence Charts graphs [51], and Petri Nets [53]. We introduce a text-book example of a multithreaded program, *producer-consumer* in Example 1.

Example 1 (Producer-consumer). We consider the classical *producer-consumer* example where a thread pushes items to a shared queue (generating a `produce` event), and another thread consumes items (one at a time) from the queue for processing (generating a `consume` event). We specify that consumers must not remove an item unless the queue contains one, and all items placed on the queue must be eventually consumed. Figure 1 illustrates the statements executed by two different threads: thread 0, and thread 1, representing respectively a producer and a consumer. Each statement is given a number for clarity. Both the producer and consumer use a shared queue shown in Listing 1.1. Statements in different threads can execute concurrently. We illustrate some correct and incorrect executions. Two correct executions have the following orders: ①②③④ and ①③②④; they comply with the specification. The execution with the order: ❷①③④ is incorrect, as a `consume` attempts to retrieve an element from an empty queue. The execution with only the statements: ①③② is incorrect, as there remains an element to be consumed. The execution with the order: ❷❹①③ violates both conditions in the specification, since two `consume` events happen when the queue is empty, and after the executions there are two elements left to be consumed.

---

[1] By soundness, we refer to the general principle of monitors detecting specification violation or compliance only when the actual system produces behavior that respectively violates or complies with the specification.
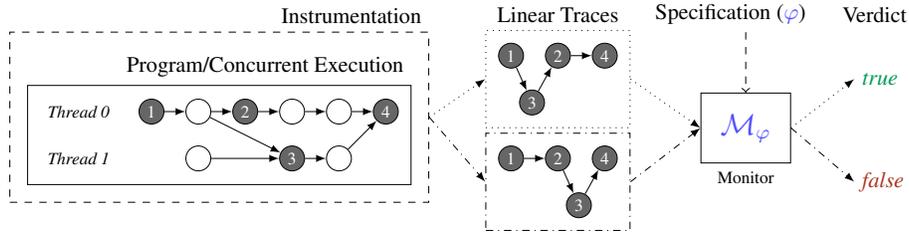
Fig. 2: RV flow and the impact of linearizing traces. Before runtime, RV is applied to a program with a concurrent execution (dashed): a monitor $\mathcal{M}_\varphi$ is synthesized from a property φ, and the program is instrumented to retrieve its relevant events. At runtime, we observe two possible linear traces that could lead to verdicts (*true* or *false*) when processed by the same monitor.

Monitoring multithreaded programs. RV has initially focused on utilizing totally ordered traces, as it uses formalisms inspired from Linear Temporal Logic (LTL) or finite-state machines as specifications [46,13,54,52], until recently with the introduction of stream-based RV [25,38,45], decentralized monitoring [11], and RV of hyperproperties [18]. Most of the top[2] existing tools for the online monitoring of Java programs rely on totally ordered traces and provide multithreaded monitoring support using one or more of the *three* modes. The first mode allows *per-thread* monitoring. The *per-thread* mode specifies that monitors are only associated with a given thread, and receive all events of the given thread. Monitors are unable to check properties that involve events across threads. This boils down to doing classical RV of single-threaded programs, assuming each thread is an independent program. When examining each thread or process while excluding others, one ignores the inter-thread dependencies, and it is therefore insufficient. For example, it is impossible to monitor *producer-consumer* illustrated in Example 1, as events happen on separate threads. In that setting, a specification cannot express behavior involving events across threads. The second mode allows for *global* monitoring. It spawns global monitors, and ensures that events are fed to the monitor *atomically*, by utilizing locks. As such, a program execution is *linearized* so that it can be processed by the monitors. Locks force events to be totally ordered across the entire execution, which oversimplifies and ignores concurrency, as illustrated in Example 2.

Example 2 (Linearization). Figure 2 illustrates the typical RV flow for some property φ with a monitor $\mathcal{M}_\varphi$, where during the execution, an instrumented parallel program feeds a trace to a monitor. Filled circles represent the events relevant to the RV specification, and are numbered simply to distinguish them. We notice that, in the case of a concurrent execution, the trace could differ based on the linearization strategy which influences the observation order. In the first trace, event 3 precedes event 2, while in the second trace, we have the opposite. This could potentially impact the verdict of the monitor if the specification relies on the order between events 2 and 3. We recall *producer-consumer* from Example 1: if the program is not properly synchronized, linearizing the concurrent events could lead two different traces: ①②③④, and ❷①③④. The first trace complies with the specification while the second violates it.

―――――――――
[2] Based on the first three editions of the Competition on Runtime Verification [6,32,55,8].

The third mode allows monitors to receive events concurrently. This is typically done by providing a flag *unsynchronized*. In this mode, practitioners should handle the concurrency on their own, and in some cases specify their own monitoring logic. Writing additional concurrency logic, and managing concurrency has *three* disadvantages. First, by writing the monitors manually, we defeat the purpose of automatically generating monitors from a given formalism. Second, the manual monitors created may miss key information needed for managing concurrency. This extra information may require to implement additional instrumentation outside the tool. Third, the process is complicated due to concurrency, and is error-prone. We elaborate on the complications in Sec. 4. As such, we first ask if monitors are to be generated from a formalism.

Q0 *Is the developer using the tool to automatically generate monitor logic?*

For the scope of this tutorial, we focus on the formalisms from which monitors could be synthesized. As such, we consider the answer to **Q0** is *yes*.

Overview. In this tutorial, we explore RV tools that explicitly handle multithreaded programs. We illustrate the problem of monitoring a parallel program using existing techniques. In doing so, we overview the related approaches, some of the existing tools, and their shortcomings. We discuss their assumptions, advantages, limitations, and suitability when tackling two textbook parallel programs: producer-consumer and readers-writers. In particular, we use manually written monitors using AspectJ [43,58], Java-MOP [16,17,52], and RVPredict [42] to explore the challenges to monitoring multithreaded programs. Overall, the challenges of monitoring multithreaded programs stem from the following facts:

- events in a concurrent program follow a partial order;
- most formalisms used by RV do not account for partial orders, but specify behavior over sequences of events (i.e., events are totally ordered); and
- an instrumented program must capture the order of events as it happens during the execution to pass it to monitors.

Moreover, we explore the situations where:

- a linear trace does not represent the underlying program execution;
- a linear trace hides some implicit assumptions which affect RV; and
- it is insufficient to use a linear trace for monitoring multithreaded programs.

By analyzing the interplay between specification formalisms and concurrent executions of programs, we propose *four* questions RV practitioners may ask themselves to classify and determine the situations in which it is reliable to use the existing tools and approaches as well as the situations where we believe more work is needed.

An online version of the tutorial [30] is provided with the programs, tools, and an interactive guide to reproduce and experiment with the examples provided in the tutorial. The examples included in the online tutorial are marked in the rest of the paper with the dagger sign (†).

2   Exploring Tools and Their Supported Formal Specifications

Runtime Verification approaches typically automatically synthesize monitors by relying on a formal specification of the expected behavior. A specification formalism allows to express properties that partition the system behaviors into correct and incorrect ones. As such, for a multithreaded program, we must first check the available properties that we can verify. We first classify the various approaches by considering the specification formalism alone.

2.1   Approaches Relying on Total-Order Formalisms

The first pool consists of tools and approaches where the specification language itself relies on a total order of events, as the input to monitors consists of words. We consider the tools commonly used for RV using those found in the RV competitions [8,32,55].

Java-MOP. Java-MOP [16,17,52] follows the design principle that specifications and programs should be developed together. Java-MOP provides *logic plugins* to express the specifications in several formalisms. Logic plugins include: finite-state machines, extended regular expressions, context-free grammars, past-time linear temporal logic, and string rewriting systems.

Tracematches. Tracematches [4,13] is another approach that uses regular expressions over user-specified events as specifications. Tracematches defines points in the execution where events occur, and specifies the actions to execute upon matching. Tracematches considers the semantics of such matching on large programs or multiple program runs, while binding the context associated with each event to the sequence. For example, it considers when a pattern matches multiple times, or matches multiple points in the program.

MarQ. MarQ [54] is designed for monitoring properties expressed as Quantified Event Automata (QEA) [5]. MarQ focuses on performing highly optimized monitoring, by providing full control of monitors lifecycles and garbage collection. Furthermore, it introduces quantification and distinguishes quantified from free variables in a specification, this allows finer control over the monitoring procedure by managing the replication of monitoring (slicing). MarQ relies on the developer to instrument the program with AspectJ to send the events to the QEA.

LARVA. LARVA [22] uses dynamic automata with timers and events [21]. LARVA focuses on monitoring real-time systems where timing is of importance. LARVA specifications feature timeouts and stopwatches. LARVA is also capable of verifying large programs by storing events in a database and allowing the monitors to "catch up" with the system as it executes [20].

Remark 1 (Unsynchronized monitors). While we focus on formalisms capable of automatically generating monitors, we note that it is still possible to write unsynchronized monitors manually. We explain in Sec. 4 the difficulties that make the process error-prone. Java-MOP provides the *unsynchronized* flag to specify that no additional locks should be added, thus allowing monitors to receive events concurrently. *Logic plugins* would no longer be used to automatically synthesize monitors. MarQ by default is not thread safe [54]. The developer must pre-process the events before passing them to the QEA monitor.

2.2   Approaches Focusing on Detecting Concurrency Errors

The second pool of tools is concerned with specific behavior for concurrent programs. We consider absence of data races and deadlock freedom. Tools used that can verify specific properties related to concurrency errors include RVPredict [42] and Java PathExplorer (JPaX) [39]. Further discussion on concurrency errors and additional tools are discussed in [48].

RVPredict. RVPredict relies on Predictive Trace Analysis (PTA) [42,57]. PTA approaches model the program execution as a set of traces corresponding to the different orderings of a trace. As such, they encode the trace minimally, then restrict the set of valid permutations based on the model that is allowed. The approach in [42] describes a general sound and complete model to detect data races in multithreaded programs and implement it in RVPredict. Traces are ordered permutations containing both control flow operations and memory accesses, and are constrained by axioms tailored to data race and sequential consistency. While [42] can, in theory, model behavioral properties, RVPredict monitors only data races, but does so very efficiently.

JPaX. Similar to RVPredict, Java PathExplorer (JPaX) [39] is a Java tool designed for multithreaded programs. JPaX uses bytecode-level instrumentation to detect both race conditions and deadlocks in a multithreaded program execution. To do so, JPaX tracks information on locks and variables accessed by various threads during an execution. JPaX supports standard formalisms such as LTL and finite-state machines. However, it separates those from the two mentioned concurrency properties, and defaults to providing an event stream to the monitors similar to automata-based approaches.

2.3   Approaches Utilizing Multiple Traces

The third pool consists of approaches specifying behavior that spans multiple traces.

Stream-based techniques. Stream-based techniques include LOLA [25], BeepBeep [38], and more recently, the Temporal Stream-Based Specification Language [45,23,26]. Stream-based specifications rely on named streams to provide events. These streams are then aggregated using various functions that modify the timing, filter events, and output new events.

Decentralized monitoring. Decentralized monitoring considers the system as a set of components sharing a logical timestamp. It uses monitoring algorithms and communication strategies to monitor one specification over components by avoiding synchronization and with the aim of minimizing the communication costs. Algorithms manage a decentralized trace associating each event with a component and a timestamp; essentially managing for each component a totally ordered trace. DecentMon [10,19] is a tool capable of simulating the behavior of decentralized monitoring algorithms.

Decentralized specifications. Decentralized specifications [28] generalize decentralized monitoring by defining a set of monitors, additional atomic propositions that represent references to monitors, and attaches each monitor to a component. A monitor reference is evaluated as if it was an oracle. THEMIS [29] is a tool capable of monitoring decentralized specifications.

Hyperproperties. *Hyperproperties* [18] are specified over sets of traces. Typically, hyperproperties make use of variables that are quantified over multiple traces. RV approaches have been implemented to verify hyperproperties using rewriting [14], and using model checking and automata [34]. RVHyper [33] is a tool capable of verifying hyperproperties on sets of traces.

2.4   Outcome: A First Classification

Since concurrent executions exhibit a partial order between events, formalisms that rely on total order require that the partial order be coerced into a total order. Our first consideration for monitoring concurrent programs relies solely on the specification formalism.

Q1 *Are the models of the specification formalism based on a total order?*

If the answer to **Q 1** is *yes*, then we are concerned with the first pool of tools. We elaborate on further considerations for total order approaches in Sec. 3. Otherwise, we verify whether or not we are checking very specific properties on partial orders, such as data race or deadlock freedom.

Q2 *Are we only concerned with the absence of data races or deadlock freedom?*

If the answer to **Q 2** is *yes*, then we are concerned with the second pool of tools, keeping in mind that they are unable to handle arbitrary specifications. Otherwise, we are concerned with the third pool, we elaborate on the potential of using these approaches in Sec. 5.2.

3   Linear Specifications for Concurrent Programs

In this section, we are concerned with RV approaches that rely on total-order formalisms (e.g., automata, LTL, regular expressions). We refer to specifications that use total-order formalisms to describe the behavior of the system as *linear specifications*. We explore the assumptions and outcomes of checking properties specifying total-order behavior.

3.1   Per-thread Monitoring

Overview. A simple approach to monitor multithreaded programs is to consider each thread in the program execution independent. That is, the monitoring technique assumes that each thread is a separate serial program to monitor. A monitor is assigned to each thread and receives only events pertaining to that thread. This is called *per-thread* monitoring. Java-MOP and Tracematches support flag *perthread* [4,35] to monitor a property on each thread independently. It is also possible to use MarQ by quantifying over the threads, to monitor each thread independently for a given property.

Example 3 (Per-thread iterator[†]). We use for example the classical property described in [16] "An iterator's method hasNext() must always be called at least once before a call to method next()". Monitoring *per-thread* proves useful, when we are concerned about the usage of iterators in a given thread, and not across threads. Using Java-MOP, we can monitor a simple program that has two threads processing a shared list of integers concurrently. Each thread creates an iterator on the shared list, the first finds the minimum, while the second finds the maximum. In this case, it is sufficient to check that the iterator usage is correct for each thread independently.

Limitations. Since *per-thread* monitoring performs RV on a single thread, and all events in a given thread are totally ordered, it follows that monitoring is sound in such situations. However, in most cases, we may be interested in monitoring events across threads. This is the case with *producer-consumer* detailed in Example 1. To monitor the program we need to keep track of produces and consumes. By considering threads separately, one is not able at all to monitor the correct behavior, as producer and consumer are separate threads. Monitoring *per-thread* is not useful in this setting. Therefore, it becomes important to distinguish between properties for which events are shared across threads.

Q3 *Does there exist a model of the specification where events are generated by more than a single thread?*

We addressed in this section the tools and limitations when the answer to **Q3** is *no*. When the answer to **Q3** is *yes*, a developer has to consider *global* monitoring, explained in Sec. 3.2.

3.2   Global Monitoring

Overview. Whenever the specification formalism relies on events across threads, the existing approaches that use a total-order formalism typically define global monitors. This is the default mode for Java-MOP, MarQ, and for Tracematches this is called "*global tracematch*". This is the only mode for LARVA. Furthermore, these tools typically include synchronization guards on such monitors. For example, LARVA synchronizes events passed to the monitors, such that a monitor cannot receive two events concurrently, while MarQ requires the developer to specify synchronization when needed, and Java-MOP offers an *unsynchronized* flag, to disable locking on monitors.

   We discussed the implications of using *unsynchronized* in Sec. 1.

Example 4 (Monitoring producer-consumer[†]). We monitor *producer-consumer* (Example 1) using Java-MOP, LARVA, and MarQ[3]. The property can be expressed as a context-free grammar (CFG) using the rule: `S -> S produce S consume | epsilon`[4]. We specify the property for each tool[4] and associate events `produce` and `consume` with adding and removing elements from a shared queue, respectively. We first verify this example using *per-thread* monitoring using Java-MOP, and notice quickly that the property is violated, as the first monitor is only capable of seeing produces, and the second only consumes. Using global monitoring, we monitor a large number of executions (10,000) of two variants of the program, and show the result in Table 1. For each execution, the producer generates a total of 8 `produce` events, which are then processed using up to 2 consumers. The first variant is a correctly synchronized *producer-consumer*, where locks ensure the atomic execution of each event. The second variant is a non-synchronized *producer-consumer*, and allows the two events to be fed to the monitors concurrently. In both cases, the monitor is synchronized to ensure that the monitor processes each event atomically. Additional locks are included by Java-MOP and LARVA, we introduce a lock for MarQ, as it is not thread-safe. This is consistent

---

[3] On Java openjdk 1.8.0_172, using Java-MOP version 4.2, MarQ version 1.1 commit 9c2ecb4 (April 7, 2016), and LARVA commit 07539a7 (Apr 16, 2018).

[4] Equivalent monitors and specifications for each tools can be found in Appendix A.

Table 1: Monitoring 10,000 executions of 2 variants of *producer-consumer* using global monitors. Reference (REF) indicates the original program. Column **V** indicates the variant of the program. Column **Advice** indicates intercepting *after* (A) and *before* (B) the function call, respectively. Columns **True** and **False** indicate the number of executions (#) and the percentage over the total number of executions (%) for which the tool reported these verdicts.

V	Consumers	Tool	Advice	True		False		Timeout	
				#	%	#	%	#	%
1	1-2	REF			-			0	(0%)
		JMOP	A	10,000	(100%)	0	(0%)	0	(0%)
			B	10,000	(100%)	0	(0%)	0	(0%)
		MarQ	A	10,000	(100%)	0	(0%)	0	(0%)
			B	10,000	(100%)	0	(0%)	0	(0%)
		LARVA	A	10,000	(100%)	0	(0%)	0	(0%)
			B	10,000	(100%)	0	(0%)	0	(0%)
2	1	REF			-			631	(6.3%)
		JMOP	A	4,043	(40.43%)	5,957	(59.57%)	0	(0%)
			B	7,175	(71.75%)	6	(0.06%)	2,819	(28.19%)
		MarQ	A	4,404	(44.04%)	5,583	(55.83%)	13	(0.13%)
			B	9,973	(99.73%)	16	(0.16%)	11	(0.11%)
		LARVA	A	4,755	(47.55%)	5,245	(52.45%)	0	(0%)
			B	9,988	(99.88%)	2	(0.02%)	10	(0.10%)
2	2	REF			-			4,785	(47.85%)
		JMOP	A	128	(1.28%)	9,220	(92.20%)	652	(6.52%)
			B	1,260	(12.60%)	7,617	(76.17%)	1,123	(11.23%)
		MarQ	A	33	(0.33%)	9,957	(99.57%)	10	(0.10%)
			B	432	(4.32%)	9,530	(95.30%)	38	(0.38%)
		LARVA	A	250	(2.50%)	9,488	(94.88%)	262	(2.62%)
			B	5,823	(58.23%)	4,131	(41.31%)	46	(0.46%)

as to check the CFG (or the automaton for LARVA and MarQ), we require a totally ordered word, as such traces are eventually linearized.

In the first variant, the monitor outputs verdict *true* for all executions. This is consistent with the expected behavior as the program is correctly synchronized, as such it behaves as if it were totally ordered. However, with no proper synchronization, `produce` and `consume` happen concurrently, we obtain one of two possible traces:

$$tr_1 = \texttt{produce} \cdot \texttt{consume} \quad \text{and} \quad tr_2 = \texttt{consume} \cdot \texttt{produce}.$$

While tr_1 seems correct and tr_2 incorrect, `produce` and `consume` happen concurrently. After doing 10,000 executions of the second variant, monitoring is unreliable: we observe verdict *true* for some executions, while for others, we observe verdict *false*. Even for the same tool, and the same number of consumers, we notice that the reported verdicts vary depending on whether or not we choose to intercept before or after the function call to create the event. For example, even when using a single consumer with
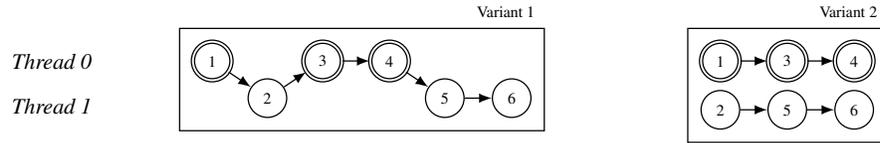
Fig. 3: Concurrent execution fragments of *producer-consumer* variants. Double circle: `produce`, normal: `consume`. Events are numbered to distinguish them.

Java-MOP, we see that the verdict rate for verdict *false* goes down from 60% when intercepting before the function call, to almost 0% when intercepting after the function call. We note that selecting to intercept before or after a method call can depend on the specification. For consistency reasons, we chose to intercept both events in the same way. Either choice produces inconsistent verdicts when concurrency is present, due to context switches.

In the second variant, the consumer must check that the queue has an element, and then poll it to recover it. Since it is badly synchronized, it is possible to deadlock as the check and the poll are not atomic. In this case, the program cannot terminate. To distinguish deadlocked executions, we terminate the execution after 1 second, and consider it a timeout, since a non-deadlocked execution takes less than 10 milliseconds to execute. It is important to note that when the specification detects a violation the execution is stopped, this could potentially lower the rate of timeouts. The rate of timeout of the original program (REF) is given as reference. We notice that the tools interfere with the concurrency behavior of the program in two ways. First, the locking introduced by the global monitoring can actually force a schedule on the program. We observe that when a single consumer is used and locks are used before the function call. In this case, the rate of getting verdict *true* is higher than when introduced after the call (72% for Java-MOP, 99.7% for MarQ, and 99.8% for LARVA). When the locks are applied naively, they can indeed correct the behavior of the program, as they force a schedule on the actions `produce` and `consume`. This, of course, is coincidental, when 2 consumers are used, we stop observing this behavior. Second, we observe that changing the interception from before to after the function call modifies the rate of timeout. For example, when using 1 consumer, the reference rate is 6% (REF). When using Java-MOP (B), the rate goes up to 28%, while for LARVA (B) it goes down to 0.1%. It is possible to compare the rate of timeout of Java-MOP (B) and LARVA (B) since the monitor is not forcing the process to exit early, as the rate of reaching verdict *false* is low for both ($<$ 0.1%). We elaborate more on the effect of instrumentation on concurrency in Sec. 4.

To understand the inconsistency in the verdicts, we look at the execution fragments of each variant in Fig. 3. In the first variant, the program utilizes locks to ensure the queue is accessed atomically. This allows the execution to be a total order. For the second variant, we see that while we can establish order between either `produce`, or `consume`, we cannot establish an order between events. During the execution, multiple total orders are possible, and thus different verdicts are possible.

Limitations. It is now possible to distinguish further situations where it is reliable to use global monitors. We notice that to evaluate a total order formalism, we require a trace which events are totally ordered. When dealing with a partial order, tools typically

Listing 1.2: RVPredict (partial) output for *producer-consumer* variant 2.

```
1  ─────────────Instrumented execution to record the trace─────────────
2  [RV−Predict] Log directory: /tmp/rv−predict2523508450121758452
3  [RV−Predict] Finished retransforming preloaded classes.
4  main Complete in 28
5  Data race on field java.util.LinkedList.$state:
6      Read in thread 14
7       > at SynchQueue.consume(SynchQueue.java:24)
8         at Consumer.run(Consumer.java:14)
9      Thread 14 created by thread 1
10        at java.util.concurrent.ThreadPoolExecutor.addWorker(Unknown Source)
11
12     Write in thread 13
13      > at SynchQueue.produce(SynchQueue.java:18)
14        at Producer.run(Producer.java:19)
15     Thread 13 created by thread 1
16        at java.util.concurrent.ThreadPoolExecutor.addWorker(Unknown Source)
```
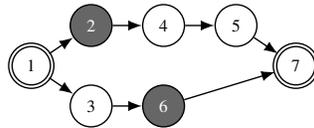

Fig. 4: Concurrent execution fragment of *1-Writer 2-Readers*. Double circle: `write`, normal: `read`. Events are numbered to distinguish them. Events 2 and 6 are an example of concurrent events as there is no order between them.

use locks and ensure that the partial order will be coerced into a total order. We see that the monitoring of the second variant failed since the program was not properly synchronized. One could assume that it is necessary to first check that the program is properly synchronized, and perhaps deadlock-free as well. To do so, one could use RVPredict or JPaX to verify the absence of data race (as shown in Example 5). Upon verifying that the program is synchronized, one could then run global monitors.

Example 5 (Detecting data race[†]). Let us consider the second variant of *producer-consumer* as described in Example 4. Listing 1.2 displays the (partial) output of executing RVPredict on the program. Particularly, we focus on one data race report (out of 4). We notice that in this case, lines 7 and 13 indicate that the data race occurs during those function calls. Yet, these are the calls we used to specify the `produce` and `consume` events. In this case, we can see that the data race occurs at the level of the events we specified. Upon running RVPredict on the first variant, it reports no data races, as it is properly synchronized.

While checking the absence of data race is useful for the case of *producer-consumer*, it is not enough to consider a properly synchronized program to be safe when using global monitors. This is due to the possible existence of concurrent regions independently from data race. We illustrate the case of concurrent regions in Example 6.

Example 6 (1-Writer 2-Readers[†]). Figure 4 illustrates a concurrent execution fragment of *1-Writer 2-Readers*, where a thread can write to a shared variable, and two other threads can read from the variable. The threads can read concurrently, but no thread can write or read while a write is occurring. In this execution, the first reader performs 3

reads (events 2, 4, and 5), while the second reader performs 2 reads (events 3 and 6). We notice that indeed, no reads happen concurrently. In this case, we see that the program is correctly synchronized (it is data-race free and deadlock-free). However, we can still end up with different total orders, as there still exists concurrent regions. By looking at the concurrent execution, we notice that we can still have events on which we can establish a total order[5].

On the one hand, a specification relying on the order of events found in concurrent regions (i.e., "*the first reader must always read before the second*") can still result in inconsistent monitoring, similarly to *producer-consumer*. On the other hand, a specification relying on events that can always be totally ordered (i.e., "*there must be at least one read between writes*") will not result in inconsistent monitoring. We notice that to distinguish these two cases, we rely (i) on the order of the execution (concurrent regions), and (ii) the events in the specification. Two events that cannot be ordered are therefore called *concurrent events*. For example, the events 2 and 6 are concurrent, as there is no order relation between them. Instrumenting the program to capture *concurrent events* may also be problematic as we will explain in Sec. 4.

3.3   Outcome: Refining the Classification

We are now able to formulate the last consideration for totally ordered formalisms.

Q4 *Is the satisfaction of the specification sensitive to the order of concurrent events?*

If the answer to **Q4** is *no*, then it is possible to linearize the trace to match the total order expressed in the specification. Otherwise, monitoring becomes *unreliable* as the concurrency can cause non-determinism, or even make it so the captured trace is not a representation of the execution as we explain in Sec. 4.

Remark 2 (Expressiveness). We noticed that utilizing linear specifications for monitoring multithreaded programs works well when the execution of the program can be reduced to a total order. On the one hand, we see *per-thread* monitoring (Sec. 3.1) restricting events to the same thread. On the other hand, we see *global* monitoring restricting the behavior to only those that can be linearized. As such, in these cases, the interplay between trace and specification constrains the expressiveness of the monitoring to either the thread itself, or the segments in the execution that can be linearized.

4   Instrumentation: Advice Atomicity

Generally, trace collection is done after instrumentation of the program using AspectJ, or other techniques (such as bytecode instrumentation). As mentioned in Sec. 1, it is still possible to specify *unsynchronized* monitors and handle concurrency without the tool support. We note that using AspectJ for instrumentation is found in Java-MOP, Tracematches, MarQ, and LARVA [8]. In this section, we show that instrumentation may lead to unreliable traces in concurrent regions.

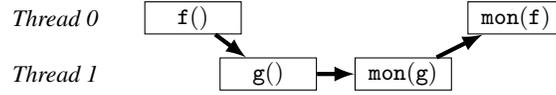[5] This is similar to the notion of *linearizability* [40].

Fig. 5: Advice execution (`mon`) with context-switches leading to incorrect trace capture.

1	g	f_trace
2	f	g_trace
3	f	f_trace
4	g	g_trace
5	f	f_trace
6	g	g_trace
7	f	f_trace
8	g	g_trace
9	f	f_trace
10	g	f_trace
11	f	g_trace
12	f	f_trace
13	g	g_trace
14	g	g_trace

(a) Comparison between the system trace (left) and the trace collected by the monitor (right).

Tool	Advice	Sync	Identical	Different
AspectJ	A	✓	4,912	5,088
	B		9,170	830
Java-MOP	A	✓	1,737	8,263
	B		9,749	251
LARVA	A	✓	8,545	1,455
	B		9,992	8
Java-MOP	A	✗	2,026	7,974
	B		9,517	483

(b) Comparing traces collected with AspectJ, LARVA, and Java-MOP across 10,000 executions. The column **Advice** indicates respectively intercepting *after* (A) and *before* (B) the function call.

Fig. 6: Comparison of collected traces using instrumentation and the system trace.

4.1 Extracting Traces

Extracting a trace from a program execution often requires executing additional code at runtime. For example, to capture a method call, one could insert a print statement before or after the method call. This extra running code is referred to as *advice* by AspectJ. When an action is executed, *the code responsible for gathering the trace will not, in general, execute atomically with the action.* For multithreaded programs, the execution order may be incorrectly captured due to context switches between threads. To illustrate the issues caused by context switches, we have two threads with a race condition on a call to function f and g respectively, we match the call and execute the advice right after the call. We show this by adding a call to the advice code `mon()`, right after the function call. We see in Fig. 5 that in the execution the call to function f precedes the call to function g, however, due to context switches, the advice associated with g (`mon(g)`) executes before that associated with function f (`mon(f)`). In this case the order perceived by the monitors is $g \cdot f$ while the order of the execution is $f \cdot g$. In this scenario, the generated trace is not representative of the execution, and thus the check performed by the monitor is unreliable.

Example 7 (Advice Atomicity[†]). For this example, we create two threads such that each calls a unique function (f and g, respectively) an equal number of times. Each function consists of a single print statement (to `stdout`) indicating the function name. We create a simple monitor that prints (to `stderr`) the same function name while appending "_trace". Then, we verify that the traces are identical, that is the prints from within the functions follow the same order as those in the monitor. Figure 6a shows a fragment of a trace that is different. We see at lines (1-2) that the trace of the monitor starts with $f \cdot g$ while the in the program execution the order is $f \cdot g$. Figure 6b shows the difference

between the captured trace by the monitor and the trace of the system, using monitors created manually with AspectJ, and automatically with Java-MOP and LARVA. The monitor created manually with AspectJ is also representative of MarQ as MarQ relies on the user writing the event matching in AspectJ, then calling the QEA monitor. Column **Sync** distinguishes the case when using *unsynchronized* in Java-MOP. We notice that the traces differ from the actual program execution for AspectJ, Java-MOP and LARVA. Traces appear to differ more when intercepting after the function call. In AspectJ, the rate of identical traces drops from 91% (B) to 49% (A). This drop is also visible for LARVA and Java-MOP. This is not surprising as Java-MOP and LARVA use AspectJ for instrumentation while introducing some variation as each tool has some additional computation performed on matching. The rate change could be associated with either the specific program or the virtual machine in this case, as the added computation from the monitors and AspectJ could affect the schedule. More importantly, we notice that even when the monitors are synchronized, the captured trace is not guaranteed to be identical to that of the execution.

This problem can only be solved if atomicity for the granularity level can be guaranteed. In general, source-level instrumentation of method calls with AspectJ, or even bytecode instrumentation at the INVOKE level will still not be atomic. Adding a lock not only increases overhead, but can also introduce deadlocks if the method invocation is external to the code being instrumented (e.g., calls to libraries). However, by adding locks one can modify the behavior of the program as illustrated in Example 7, as such one needs to minimize the area to which the lock is applied.

4.2   Discussion

In certain conditions, capturing traces can still be done in the case of concurrent events. First, a developer must have full knowledge of the program (i.e., it must be seen as a *white box*), this allows the developer to manually instrument the locks to ensure atomic capture, avoiding deadlocks and managing external function calls carefully. Second, we require that the instrumented areas tolerate the interference, and therefore must prove that the interference does not impact significantly the behavior of the program, by modifying the schedule. In this case, one could see that global monitoring (Sec. 3.2) reports correct verdicts *for the single execution*.

Remark 3 (Monitor placement). An additional important aspect for tools pertains to whether the monitors are inlined in the program or execute separately. For multithreaded programs, instrumentation can place monitors so that they execute in the thread that triggers the event, or in a separate thread, or even process. These constitute important implementation details that could limit or interfere with the program differently. However, for the scope of this paper, we focus on issues that are relevant for event orders and concurrency.

5   Reasoning About Concurrency

Section 3 shows that approaches relying on total order formalisms are only capable of reliably monitoring a multithreaded program when the execution boils down to a total order. Therefore, it is important to reason about concurrency when designing monitoring tools, while still allowing behavioral properties. We present GPredict [41] in

Sec. 5.1, a concurrency analysis tool that can be used for specifying behavior over concurrent regions. We discuss in Sec. 5.2 the potential of multitrace approaches, first introduced in Sec. 2.3. In Sec. 5.3, we present certain approaches from outside RV that may prove interesting and provide additional insight.

5.1 Generic Predictive Concurrency Analysis

Concurrent behavior as logical constraints solving. The more general theory behind RVPredict (Sec. 2.2) develops a sound and maximal causal model to analyze concurrency in a multithreaded program [42]. In this model, the correct behavior of a program is modeled as a set of logical constraints, thus restricting the possible traces to consider. The theory supports any logical constraints to determine correctness, it is possible to encode a specification on multithreaded programs as a set of logical constraints. However, allowing for arbitrary specifications to be encoded while supports in the model, is not supported in the provided tool (RVPredict).

GPredict. Using the same sound and maximal model for predictive trace analysis [42] discussed in Sec. 2.2, GPredict [41] extends the specification formalism past data-races to behavior. Specifications are able to include behavioral, user-specified events, and are extended with thread identifiers, atomic regions, and concurrency. Events are defined similarly to Java-MOP using AspectJ for instrumentation. Atomic regions are special events that denote either the start or end of an atomic region. Each atomic region is given an ID. The specification formalism uses regular expressions extended with the concurrency operator "||" which allows events to happen in parallel.

Example 8 (Specifying concurrency). Listing 1.3 shows a specification for GPredict written for a multithreaded program, we re-use the example from [41]. The program consists of a method (m) of an object which reads and writes to a variable (s). Lines 2 and 5 specify the events that denote respectively reaching the start and end of method (m). Line 3 and 4 specify respectively the `read` and `write` events. Lines 7 and 8 illustrate respectively specifications for atomic regions and concurrency. The events in the specification can be parametrized by the thread identifier, and a region delimiter. To specify an atomic regions, an event can indicate whether it is the start or end of a region using the characters > and < respectively. The delimiter is followed by a region identifier, which is used to distinguish regions in the specification. In this case, we see that the `begin` and `end` events emitted by thread `t1` delimit an atomic regions in which a read by thread `t1` must be followed by a write by thread `t2`, which is followed by a write by thread `t1`. The specification is violated if any of the events happen in a different order or concurrently. To specify concurrent events, one must utilize "||" as shown on Line 8. In this case, the specification says that a read in thread `t1` can happen in parallel with a write in thread `t2`.

Limitations. While GPredict presents a general approach to reason about behavioral properties in concurrent executions, and hence constitutes a solution to monitoring when concurrency is present, it still requires additional improvements for higher expressiveness and usability. Notably, GPredict requires specifying thread identifiers explicitly in the specification. This requires specifications with multiple threads to become extremely verbose, and cannot handle a dynamic number of threads. For example, in the

Listing 1.3: GPredict specification depicting atomic regions.

```
1  AtomicityViolation (Object o){
2    event begin before(Object o) : execution(m());
3    event read  before(Object o) : get(* s) && target(o);
4    event write before(Object o) : set(* s) && target(o);
5    event end   after(Object o)  : execution(m());
6
7    pattern: begin(t1, <r1) read(t1) write(t2) write(t1) end(t1,>r1)
8  //pattern: read(t1) || write(t2)
9  }
```

case of *readers-writers*, adding extra readers or writers requires rewriting the specification and combining events to specify each new thread. The approach behinds GPredict can also be extended to become more expressive, e.g. to support counting events to account for fairness in a concurrent setting.

5.2   Multi-trace Specifications: Possible Candidates?

RV approaches and tools that utilize multiple traces include approaches that rely on streams, decentralized specifications, and hyperproperties (as described in Sec. 2.3).

Thread events as streams. Stream-based RV techniques deal with synchronized streams in general, the order of the events is generally total. It is possible to imagine that ordering could be performed by certain functions that aggregate streams. For example, it is possible to create a stream per event per thread, and then aggregate them appropriately to handle the partial order specifications. However, as is, either specifying or adding streams to multithreaded programs remains unclear, but presents an interesting possible future direction.

Thread-level specifications as references. Decentralized specifications present various manners to implicitly deal with threads, but do not in particular deal with multithreaded programs. Since monitors are merely references, and references can be evaluated as oracles at any point during the execution. Monitors are triggered to start monitoring, and are required to eventually return an evaluation of a property. Even when specifications are totally ordered, in the sense that they are automata-based, the semantics that allow for eventual evaluation of monitors make it so monitors on threads can evaluate local specifications and explicitly communicate with other threads for the additional information.

Concurrent executions as multiple serial executions. Hyperproperties are properties defined on a set of traces. Generally used for security, they allow for instance to check different executions of the same program from multiple access levels. By executing a concurrent program multiple times, we can obtain various totally ordered traces depending on the concurrent regions. As such, a possible future direction could explore how to express concurrency specifications as hyperproperties, and the feasibility of verifying a large set of totally ordered traces.

⚠ : Non-determinism and trace collection issues (Sec. 4).
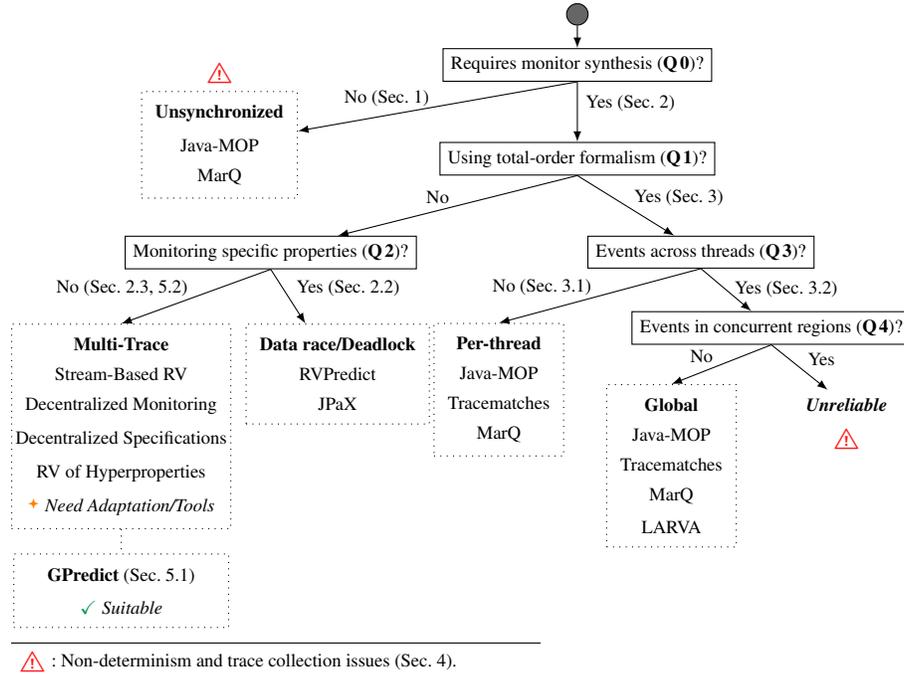
Fig. 7: RV approaches and considerations for monitoring multithreaded programs.

5.3   Inspiration From Outside RV

Detecting concurrency errors. Other approaches similar to RVPredict (Sec. 2.2) perform automatic verification and fence inference under relaxed memory models [12,44]. This ensures the correct execution of concurrent programs, but relies on static analysis and does not collect a runtime trace. Fence inference can be seen as determining concurrency segments in a program of interest with respect to the memory operations.

Relying on heuristics. Determining exact concurrency regions is costly during execution or may interfere with the execution. An interesting direction is to utilize heuristics to determine concurrent regions. BARRACUDA [27] detects synchronization errors on GPUs by instrumenting CUDA applications and performing binary-level analysis. BARRACUDA avoids large overhead as it uses heuristics to approximate the synchronization in linear traces.

Testing schedules. PARROT [24] is a *testing* framework that explores the interleavings of possible threads to test concurrent programs. PARROT analyzes the possible schedules of threads, and forces the application to explore them, thus exposing concurrency issues. The motivation behind PARROT is the realization that certain schedules occur in low probability under very specific circumstances.

6   Conclusions

We overviewed RV approaches that support multithreaded programs. By considering the various specifications formalisms, we are able to classify the tools by looking at

whether or not they rely on total-order formalisms. We investigated the limitations of linear traces in the case of RV tools relying on formalisms that use total order, and noted the situations where linear traces lead to inconsistent verdicts. After presenting tools capable of checking specific properties, we mentioned various recent RV techniques using properties over multiple traces, and discussed their potential for monitoring multithreaded programs. Figure 7 summarizes the decisions a developer must consider when choosing RV tools for multithreaded monitoring, and the limitations of the existing approaches. We caution users of tools that using a formalism in which events are specified as a total order is not reliable when monitoring concurrent events (as we cannot reliably answer **Q 4**). It is possible to monitor multithreaded programs that exhibit concurrency using GPredict (Sec. 5.1). However, issues with writing specifications easily and expressively need to be handled. Furthermore, RV techniques capable of specifying properties over multiple traces prove to be interesting candidates to extend to monitor multithreaded programs.

A Monitors

We present the specifications used for monitoring *producer-consumer* using Java-MOP (Listing 1.4), LARVA (Listing 1.5), and MarQ (Listing 1.6). The detailed findings and description is found in Sec. 3.2. The monitors were designed for *global monitoring*, to ensure the trace is fed to the corresponding formalism as a total order. As such, for MarQ locking was needed.

Listing 1.4: Java-MOP specification and monitor for *producer-consumer*.

```
1  ProdCons () {
2    event produce before () :
3      call (* Queue.add(*))
4      && cflow(execution(* SynchQueue.produce(*)))
5      { }
6    event consume before () :
7      call (* Queue.poll())
8      && cflow(execution(* SynchQueue.consume()))
9      { }
10   cfg : S -> S produce S consume | epsilon
11   @fail {
12     System.out.println("Failed!");
13     System.exit(1);
14   }
15 }
```

Listing 1.5: LARVA specification and monitor for *producer-consumer*.

```
1  IMPORTS { import java.util.*; }
2  GLOBAL{
3    VARIABLES       { int cnt = 0;}
4    EVENTS{
5      produce() = { Queue.add()  }
6      consume() = { Queue.poll() }
7    }
8    PROPERTY prodcons {
9      STATES{
10       BAD { bad {
11           System.err.println("Failed!");
12           System.exit(1);
13       }}
14       NORMAL    { ok }
15       STARTING  { starting }
16     }
17
18     TRANSITIONS{
19       starting -> bad  [consume]
20       starting -> ok   [produce\          \cnt++;]
21       ok  -> ok        [consume\ cnt > 1  \cnt--;]
22       ok  -> starting  [consume\ cnt == 1 \cnt--;]
23       ok  -> ok        [produce\          \cnt++;]
24       ok  -> bad       [consume\ cnt == 0 \       ]
25       bad -> bad       [produce]
26       bad -> bad       [consume]
27     }
28   }
29 }
```

Listing 1.6: MarQ specification and monitor for *producer-consumer*.

```java
public aspect MarQProdCon {
  // Events
  private final int PRODUCE = 1;
  private final int CONSUME = 2;
  // Produce Counter
  private int counter = 0;
  // Monitor + Lock
  Monitor monitor;
  private Object LOCK = new Object();

  before() : // Handle Event: Produce
    call(* Queue.add(*))
    && cflow(execution(* SynchQueue.produce(*)))
  {
    synchronized(LOCK){
      check(monitor.step(PRODUCE, counter));
      counter++;
    }
  }
  before() : // Handle Event: Consume
    call(* Queue.poll())
    && cflow(execution(* SynchQueue.consume()))
  {
    synchronized(LOCK){
      check(monitor.step(CONSUME, counter));
      counter--;
    }
  }
  private void check(Verdict verdict){
    if(verdict==Verdict.FAILURE){
      System.err.println("Failed!");
      System.exit(1);
    }
  }
  // Create QEA Specification
  public void init(){
    QEABuilder b = new QEABuilder("ProdCon");
    int ticket = 1;
    b.addTransition(1, PRODUCE, new int[] {ticket},
      Assignment.increment(ticket),          1);
    b.addTransition(1, CONSUME, new int[] {ticket},
      Guard.varIsGreaterThanVal(ticket, 0),
      Assignment.decrement(ticket),          1);
    b.addTransition(1, CONSUME, new int[] {ticket},
      Guard.varIsEqualToIntVal(ticket, 0), 2);
    b.addFinalStates(1);
    monitor = MonitorFactory.create(b.make());
  }
  public MarQProdCon(){ init(); }
}
```

References

1. Adhianto, L., Banerjee, S., Fagan, M.W., Krentel, M., Marin, G., Mellor-Crummey, J.M., Tallent, N.R.: HPCTOOLKIT: tools for performance analysis of optimized parallel programs. Concurrency and Computation: Practice and Experience 22(6), 685–701 (2010)
2. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: a tutorial. Computer 29(12), 66–76 (Dec 1996)
3. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: definitions, implementation, and programming. Distributed Computing 9(1), 37–49 (Mar 1995)
4. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding Trace Matching with Free Variables to AspectJ. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. pp. 345–364. OOPSLA '05, ACM (2005)
5. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.E.: Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In: Giannakopoulou and Méry [37], pp. 68–84, https://doi.org/10.1007/978-3-642-32759-9_9
6. Bartocci, E., Bonakdarpour, B., Falcone, Y.: First international competition on software for runtime verification. In: Bonakdarpour, B., Smolka, S.A. (eds.) Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8734, pp. 1–9. Springer (2014)
7. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification - Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457. Springer (2018)
8. Bartocci, E., Falcone, Y., Bonakdarpour, B., Colombo, C., Decker, N., Havelund, K., Joshi, Y., Klaedtke, F., Milewicz, R., Reger, G., Rosu, G., Signoles, J., Thoma, D., Zalinescu, E., Zhang, Y.: First international competition on runtime verification: rules, benchmarks, tools, and final results of CRV 2014. International Journal on Software Tools for Technology Transfer (Apr 2017)
9. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci and Falcone [7], pp. 1–33
10. Bauer, A., Falcone, Y.: Decentralised LTL monitoring. Formal Methods in System Design 48(1-2), 46–93 (2016)
11. Bauer, A.K., Falcone, Y.: Decentralised LTL monitoring. In: Giannakopoulou and Méry [37], pp. 85–100
12. Bender, J., Lesani, M., Palsberg, J.: Declarative fence insertion. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 367–385. OOPSLA 2015, ACM (2015)
13. Bodden, E., Hendren, L., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative Runtime Verification with Tracematches. Journal of Logic and Computation 20(3), 707–723 (Jun 2010)
14. Brett, N., Siddique, U., Bonakdarpour, B.: Rewriting-Based Runtime Verification for Alternation-Free HyperLTL. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 77–93. Springer Berlin Heidelberg, Berlin, Heidelberg (2017)
15. Bultan, T., Sen, K. (eds.): Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017. ACM (2017)
16. Chen, F., Roşu, G.: Java-MOP: A Monitoring Oriented Programming Environment for Java. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 546–550. Lecture Notes in Computer Science, Springer (Apr 2005)
17. Chen, F., Roşu, G.: Mop: an efficient and generic runtime verification framework. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications. pp. 569–588. OOPSLA '07, ACM (2007)

18. Clarkson, M.R., Schneider, F.B.: Hyperproperties. Journal of Computer Security 18(6), 1157–1210 (2010)
19. Colombo, C., Falcone, Y.: Organising LTL monitors over distributed systems with a global clock. Formal Methods in System Design 49(1-2), 109–158 (2016)
20. Colombo, C., Pace, G.J., Abela, P.: Compensation-aware runtime monitoring. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6418, pp. 214–228. Springer (2010)
21. Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: Cofer, D., Fantechi, A. (eds.) Formal Methods for Industrial Critical Systems. pp. 135–149. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
22. Colombo, C., Pace, G.J., Schneider, G.: LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper). In: Hung, D.V., Krishnan, P. (eds.) Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009. pp. 33–37. IEEE Computer Society (2009), `https://doi.org/10.1109/SEFM.2009.13`
23. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: Tessla: Temporal stream-based specification language. CoRR abs/1808.10717 (2018)
24. Cui, H., Simsa, J., Lin, Y.H., Li, H., Blum, B., Xu, X., Yang, J., Gibson, G.A., Bryant, R.E.: Parrot: A practical runtime for deterministic, stable, and reliable threads. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. pp. 388–405. SOSP '13, ACM (2013)
25. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: 12th International Symposium on Temporal Representation and Reasoning (TIME 2005). pp. 166–174. IEEE Computer Society (2005)
26. Decker, N., Dreyer, B., Gottschling, P., Hochberger, C., Lange, A., Leucker, M., Scheffel, T., Wegener, S., Weiss, A.: Online analysis of debug trace data for embedded systems. In: 2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018. pp. 851–856. IEEE (2018)
27. Eizenberg, A., Peng, Y., Pigli, T., Mansky, W., Devietti, J.: BARRACUDA: Binary-level analysis of runtime races in CUDA programs. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 126–140. PLDI 2017, ACM (2017)
28. El-Hokayem, A., Falcone, Y.: Monitoring decentralized specifications. In: Bultan and Sen [15], pp. 125–135
29. El-Hokayem, A., Falcone, Y.: THEMIS: a tool for decentralized monitoring algorithms. In: Bultan and Sen [15], pp. 372–375
30. El-Hokayem, A., Falcone, Y.: RV for Multithreaded Programs Tutorial (2018), `https://gitlab.inria.fr/monitoring/rv-multi`
31. Falcone, Y., Havelund, K., Reger, G.: A Tutorial on Runtime Verification. In: Engineering Dependable Software Systems, pp. 141–175. IOS Press (2013)
32. Falcone, Y., Nickovic, D., Reger, G., Thoma, D.: Second international competition on runtime verification CRV 2015. In: Bartocci, E., Majumdar, R. (eds.) Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9333, pp. 405–422. Springer (2015)
33. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: RVHyper : A Runtime Verification Tool for Temporal Hyperproperties. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018,

Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10806, pp. 194–200. Springer (2018)
34. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for Model Checking HyperLTL and HyperCTL*. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 30–48. Springer (2015)
35. Formal Systems Laboratory: JavaMOP4 Syntax (2018), `http://fsl.cs.illinois.edu/index.php/JavaMOP4_Syntax`
36. Gastin, P., Kuske, D.: Uniform satisfiability problem for local temporal logics over Mazurkiewicz traces. Inf. Comput. 208(7), 797–816 (2010)
37. Giannakopoulou, D., Méry, D. (eds.): FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings, Lecture Notes in Computer Science, vol. 7436. Springer (2012)
38. Hallé, S., Khoury, R.: Event Stream Processing with BeepBeep 3. In: RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools. Kalpa Publications in Computing, vol. 3, pp. 81–88. EasyChair (2017)
39. Havelund, K., Roşu, G.: An Overview of the Runtime Verification Tool Java PathExplorer. Formal Methods in System Design 24(2), 189–215 (Mar 2004)
40. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
41. Huang, J., Luo, Q., Rosu, G.: Gpredict: Generic predictive concurrency analysis. In: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Volume 1. pp. 847–857 (2015)
42. Huang, J., Meredith, P.O., Rosu, G.: Maximal sound predictive race detection with control flow abstraction. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 337–348. PLDI '14, ACM (2014)
43. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: ECOOP 2001 - Object-Oriented Programming, 15th European Conference, 2001, Proceedings. pp. 327–353. Springer (2001)
44. Kuperstein, M., Vechev, M., Yahav, E.: Partial-coherence Abstractions for Relaxed Memory Models. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 187–198. PLDI '11, ACM (2011)
45. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: TeSSLa: runtime verification of non-synchronized real-time streams. In: Haddad, H.M., Wainwright, R.L., Chbeir, R. (eds.) Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018. pp. 1925–1933. ACM (2018)
46. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming 78(5), 293–303 (May 2009)
47. Lodaya, K., Weil, P.: Rationality in algebras with a series operation. Inf. Comput. 171(2), 269–293 (2001)
48. Lourenço, J.M., Fiedor, J., Krena, B., Vojnar, T.: Discovering concurrency errors. In: Bartocci and Falcone [7], pp. 34–60
49. Manson, J., Pugh, W., Adve, S.V.: The Java Memory Model. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 378–391. POPL '05, ACM (2005)
50. Mazurkiewicz, A.W.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986. Lecture Notes in Computer Science, vol. 255, pp. 279–324. Springer (1986)

51. Meenakshi, B., Ramanujam, R.: Reasoning about layered message passing systems. Computer Languages, Systems & Structures 30(3-4), 171–206 (2004)
52. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP runtime verification framework. STTT 14(3), 249–289 (2012)
53. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, part I. Theor. Comput. Sci. 13, 85–108 (1981)
54. Reger, G., Cruz, H.C., Rydeheard, D.E.: MarQ: Monitoring at Runtime with QEA. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9035, pp. 596–610. Springer (2015)
55. Reger, G., Hallé, S., Falcone, Y.: Third international competition on runtime verification - CRV 2016. In: Falcone, Y., Sánchez, C. (eds.) Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings. Lecture Notes in Computer Science, vol. 10012, pp. 21–37. Springer (2016)
56. Reger, G., Havelund, K.: What Is a Trace? A Runtime Verification Perspective. In: Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications. pp. 339–355. Lecture Notes in Computer Science, Springer, Cham (Oct 2016)
57. Said, M., Wang, C., Yang, Z., Sakallah, K.A.: Generating data race witnesses by an SMT-Based analysis. In: NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6617, pp. 313–327. Springer (2011)
58. The Eclipse Foundation: The AspectJ project (2018), `https://www.eclipse.org/aspectj/`