Opportunistic Monitoring of Multithreaded Programs

Chukri Soueidi, Antoine El-Hokayem, Yliès Falcone

Opportunistic Monitoring of Multithreaded Programs

Chukri Soueidi, Antoine El-Hokayem, and Yliès Falcone

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

Abstract. We introduce a generic approach for monitoring multithreaded programs online leveraging existing runtime verification (RV) techniques. In our setting, monitors are deployed to monitor specific threads and only exchange information upon reaching synchronization regions defined by the program itself. They use the opportunity of a lock in the program, to evaluate information across threads. As such, we refer to this approach as opportunistic monitoring. By using the existing synchronization, our approach reduces additional overhead and interference to synchronize at the cost of adding a delay to determine the verdict. We utilize a textbook example of readers-writers to show how opportunistic monitoring is capable of expressing specifications on concurrent regions. We also present a preliminary assessment of the overhead of our approach and compare it to classical monitoring showing that it scales particularly well with the concurrency present in the program.

1 Introduction

Guaranteeing the correctness of concurrent programs often relies on dynamic analysis and verification approaches. Some approaches target generic concurrency errors such as data races [29, 37], deadlocks [11], and atomicity violations [28, 48, 58]. Others target behavioral properties such as null-pointer dereferences [27], and typestate violations [36, 39, 56] and more generally order violations with runtime verification [43]. In this paper, we focus on the runtime monitoring of general *behavioral* properties targeting violations that cannot be traced back to classical concurrency errors.

Runtime verification (RV) [9, 24, 25, 34, 43], also known as runtime monitoring, is a lightweight formal method that allows checking whether a run of a system respects a specification. The specification formalizes a behavioral property and is written in a suitable formalism based for instance on temporal logic such as LTL or finite-state machines [1, 46]. Monitors are synthesized from the specifications, and the program is instrumented with additional code to extract events from the execution. These extracted events generate the trace, which is fed to the monitors. From the monitor perspective, the program is a black box and the trace is the sole system information provided.

To model the execution of a concurrent program, verification techniques choose their trace collection approaches differently based on the class of targeted properties. When properties require reasoning about concurrency in the program, causality must be established during trace collection to determine the *happens-before* [41] relation between events. Data race detection techniques [29, 37] for instance require the causal ordering to check for concurrent accesses to shared variables; as well as predictive approaches targeting behavioral properties such as [19, 39, 56] in order to explore other

feasible executions. Causality is best expressed as a partial order over events. Partial orders are compatible with various formalisms for the behavior of concurrent programs such as weak memory consistency models [2, 4, 47], Mazurkiewicz traces [32, 49], parallel series [44], Message Sequence Charts graphs [50], and Petri Nets [51]. However, while the program behaves non-sequentially, its observation and trace collection is sequential. Collecting partial order traces often relies on vector clock algorithms to timestamp events [3,16,48,54] and requires blocking the execution to collect synchronization actions such as locks, unlocks, reads, and writes. Hence, existing techniques that can reason on concurrent events are expensive to use in an online monitoring setup. Indeed, many of them are often intended for the design phase of the program and not in production environments (see Section 5).

Other monitoring techniques relying on total-order formalisms such as LTL and finite state machines require linear traces to be fed to the monitors. As such they immediately capture linear traces from a concurrent execution without reestablishing causality. Most of the top[1] existing tools for the online monitoring of Java programs, these include tools such as Java-MOP [18, 30] and Tracematches [5], provide multithreaded monitoring support using one or more of the following *two* modes. The *per-thread* mode specifies that monitors are only associated with a given thread, and receive all events of the given thread. This boils down to doing classical RV of single-threaded programs, assuming each thread is an independent program. In this case, monitors are unable to check properties that involve events across threads. The *global* monitoring mode spawns a global monitor and ensures that the events from different threads are fed to a central monitor atomically, by utilizing locks, to avoid data races. As such, the monitored program execution is *linearized* so that it can be processed by the monitors. In addition to introducing additional synchronization between threads inhibiting parallelism, this monitoring mode forces events of interest to be totally ordered across the entire execution, which oversimplifies and ignores concurrency.

Figure 1 illustrates a high-level view of a concurrent execution fragment of *1-Writer 2-Readers*, where a writer thread writes to a shared variable, and two other reader threads read from it. The reader threads share the same lock and can read concurrently once one of them acquires it, but no thread can write nor read while a write is occurring. We only depict the read/write events and omit lock acquires and releases for brevity. In this execution, the writer acquires the lock first and writes (event 1), then after one of the reader threads acquires the lock, they both concurrently read. The first reader performs 3 reads (events 2, 4, and 5), while the second reader performs 2 reads (events 3 and 6), after that the writer acquires the lock and writes again (event 7). A user

[1] Based on the first three editions of the Competition on Runtime Verification [7, 8, 26, 53].

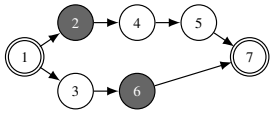

Fig. 1: Execution fragment of *1-Writer 2-Readers*. Double circle: `write`, normal: `read`. Numbers distinguish events. Events 2 and 6 (shaded) are example concurrent events.

may be interested in the following behavioral property: *"Whenever a writer performs a write, all readers must at least perform one read before the next write"*. Note that the execution here has no data races nor a deadlock, and techniques focusing on generic concurrency properties are not suitable for the property. Monitoring of this (partial) concurrent execution with both previously mentioned modes presents restrictions. For *per-thread* monitoring, since each of the readers is a thread, and the writer itself is a thread, it cannot check any specification that refers to an interaction between them. For *global* monitoring, it imposes an additional lock operation to send each read event to the monitor, introducing additional synchronization and suppressing the concurrency of the program.

A central observation we made is that when the program is free from generic concurrency errors such as data races and atomicity violations, a monitoring approach can be opportunistic and utilize the available synchronization in the program to reason about high-level behavioral properties. In the previous example, we know that reads and writes are guarded by a lock and do not execute concurrently (assuming we checked for data races). We also know that the relative ordering of the reads between themselves is not important to the property as we are only interested in counting that they all read the latest write. As such, instead of blocking the execution at each of the 7 events to safely invoke a global monitor and check for the property, we can have thread-local observations and only invoke the global monitor once either one of the readers acquires the lock or when the writer acquires it (only 3 events). As such, in this paper, we propose an approach to opportunistic runtime verification. We aim to (i) provide an approach that enables users to arbitrarily reason about concurrency fragments in the program, (ii) be able to monitor properties *online* without the need to record the execution, (iii) utilize the existing tools and formalism prevalent in the RV community, and (iv) do so efficiently without imposing additional synchronization.

We see our contributions as follows. We present a generic approach to monitor lock-based multithreaded programs that enable the re-use of the existing tools and approaches by bridging *per-thread* and *global* monitoring. Our approach consists of a two-level monitoring technique where at both levels existing tools can be employed. At the first level, a thread-local specification checks a given property on the thread itself, where events are totally ordered. At the second level, we define *scopes* which delimit concurrency regions. Scopes rely on operations in the program guaranteed to follow a total order. The guarantee is ensured by the platform itself, either the program model, the execution engine (JVM in our case), or the compiler. We assume that scopes execute atomically at runtime. Upon reaching the totally ordered operations, a scope monitor utilizes the result of all thread-local monitors executed in the concurrent region to construct a scope state, and perform monitoring on a sequence of such states. Our approach can be seen as a combination of performing global monitoring at the level of scope (for our example, we utilize lock acquires) and per-thread monitoring for active threads in the scope. Thus, we allow per-thread monitors to communicate their results when the program synchronizes. This approach relies on existing ordered operations in the program. However, it incurs minimal interference and overhead as it does not add additional synchronization, namely locks, between threads in order to collect a trace.

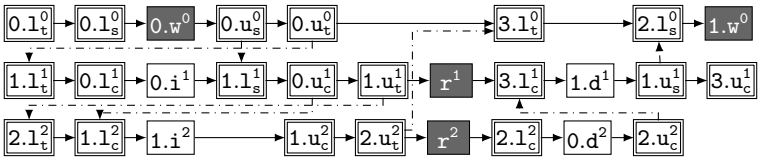

Fig. 2: Concurrent execution fragment of 1-Writer 2-Readers. Labels $\mathrm{l}, \mathrm{u}, \mathrm{w}, \mathrm{r}$ indicate respectively: lock, unlock, write, read. Actions with a double border indicate actions of locks. The read and write actions are filled to highlight them.

2 Modeling the Program Execution

We are concerned with an abstraction of a concurrent execution, we focus on a model that can be useful for monitoring the behavioral properties. We choose the smallest observable execution step done by a program and refer to it as an *action*; for instance a method call or write operation.

Definition 1 (Action). *An action is a tuple* $\langle \mathrm{lbl}, \mathrm{id}, \mathrm{ctx} \rangle$*, where:* lbl *is a label,* id *is a unique identifier, and* ctx *is the context of the action.*

The label captures an instruction name, function name, or specific task information depending on the granularity of actions. Since the action is a runtime object, we use id to distinguish two executions of the same syntactic element. Finally, the context (ctx) is a set containing dynamic contexts such as a thread identifier ($\mathrm{threadid}$), process identifier (pid), resource identifier (resid), or a memory address. We use the notation $\texttt{id.lbl}^{\texttt{threadid}}_{\texttt{resid}}$ to denote an action, omit `resid` when absent, and `id` when there is no ambiguity. Furthermore, we use the notation $\mathrm{a.threadid}$ for a given action a to retrieve the thread identifier in the context, and $\mathrm{a.ctx}(key)$ to retrieve any element in the context associated with key.

Definition 2 (Concurrent Execution). *A concurrent execution is a partially ordered set of actions, that is a pair* $\langle \mathbb{A}, \rightarrow \rangle$*, where* $\mathbb{A}$ *is a set of actions and* $\rightarrow \subseteq \mathbb{A} \times \mathbb{A}$ *is a partial order over* $\mathbb{A}$*.*

Two actions a_1 and a_2 are related (i.e., $\langle a_1, a_2 \rangle \in \rightarrow$) if a_1 happens before a_2.

Example 1 (Concurrent fragment for 1-Writer 2-Readers.). Figure 2 shows another concurrent execution fragment for *1-Writer 2-Readers* introduced in Sec. 1. The concurrent execution fragment contains all actions performed by all threads, along with the partial order inferred from the synchronization actions such as locks and unlocks (depicted with dashed boxes). Recall that a lock action on a resource synchronizes with the latest unlock if it exists. This synchronization is depicted by the dashed arrows. We have three locks: test for readers (t), service (s), and readers counter (c). Lock t checks if any reader is currently reading, and this lock gives preference to writers. Lock s is used to regulate access to the shared resource, it can be either obtained by readers or one writer. Lock c is used to regulate access to the readers counter, it only synchronizes readers. In this concurrent execution, first, the writer thread acquires the lock and writes

on a shared variable whose resource identifier is omitted for brevity. Second, the readers acquire the lock s and perform a read on the same variable. Third, the writer performs a second write on the variable.

In RV, we often do not capture the entire concurrent execution but are interested in gathering a *trace* of the relevant parts of it. In our approach, a trace is also a concurrent execution defined over a subset of actions. Since the trace is the input to any RV technique, we are interested in relating a trace to the concurrent execution, while focusing on a subset of actions. For this purpose, we introduce the notions of *soundness* and *faithfulness*. We first define the notion of *trace soundness*. Informally, a concurrent execution is a sound trace if it does not provide false information about the execution.

Definition 3 (Trace Soundness). *A concurrent trace* $tr = \langle \mathbb{A}_{tr}, \rightarrow_{\mathrm{tr}} \rangle$ *is said to be a sound trace of a concurrent execution* $e = \langle \mathbb{A}, \rightarrow \rangle$ *(written* $\mathrm{snd}(e, tr)$*) iff (i)* $\mathbb{A}_{tr} \subseteq \mathbb{A}$ *and (ii)* $\rightarrow_{\mathrm{tr}} \subseteq \rightarrow$.

Intuitively, to be sound, a trace (i) should not capture an action not found in the execution, and (ii) should not relate actions that are unrelated in the execution. While a sound trace provides no incorrect information on the order, it can still be missing information about the order. In this case, we want to also express the ability of a trace to capture all relevant order information. Informally, a *faithful trace* contains all information on the order of events that occurred in the program execution.

Definition 4 (Trace Faithfulness). *A concurrent trace* $tr = \langle \mathbb{A}_{tr}, \rightarrow_{\mathrm{tr}} \rangle$ *is said to be faithful to a concurrent execution* $e = \langle \mathbb{A}, \rightarrow \rangle$ *(written* $\mathrm{faith}(e, tr)$*) iff* $\rightarrow_{\mathrm{tr}} \supseteq (\rightarrow \cap\, \mathbb{A}_{tr} \times \mathbb{A}_{tr})$.

3 Opportunistic Monitoring

We start with distinguishing threads and events from the execution. We then define scopes that allow us to reason about properties over concurrent regions. We then devise a generic approach to evaluate scope properties and perform monitoring.

3.1 Managing Dynamic Threads and Events

Threads are typically created at runtime and have a unique identifier. We denote the set of all thread ids by TID. They are subject to change from one execution to another, and it is not known in advance how many threads will be spawned during the execution. As such, it is important to design specifications that can handle threads dynamically.

Distinguishing Threads To allow for a dynamic number of threads, we first denote thread types $\mathbb{T}$, to distinguish threads that are relevant to the specification. For example, the set of thread types for *readers-writers* is $\mathbb{T}_{\mathrm{rw}} = \{\mathrm{reader}, \mathrm{writer}\}$. By using thread types, we can define properties for specific types regardless of the number of threads spawned for a given type. In order to assign a type to a thread in practice, we distinguish a set of actions $\mathbb{S} \subseteq \mathbb{A}$ called "spawn" actions. For example in *readers-writers*, we

can assign the spawn action of a reader (resp. writer) to be the method invocation of `Reader.run` (`Writer.run`). Function $\mathrm{spawn} : \mathbb{S} \to \mathbb{T}$, assigns a thread type to a spawn action. The threads that match a given type are determined based on the spawn action(s) present during the execution. We note that a thread can have multiple types. To reference all threads assigned a given type, we use function $\mathrm{pool} : \mathbb{T} \to 2^{\mathrm{TID}}$. That is, given a type t, a thread with *threadid* tid, we have $tid \in \mathrm{pool}(t)$ iff $\exists a \in \mathbb{S} : \mathrm{spawn}(a) = t \wedge a.\mathrm{threadid} = tid$. This allows a thread to have multiple types so that properties operate on different events in the same thread.

Events As properties are defined over events, actions are typically abstracted into events. As such, we define for each thread type $\mathrm{t} \in \mathbb{T}$, the alphabet of events: $\mathbb{E}_t$. Set $\mathbb{E}_t$ contains all the events that can be generated from actions for the particular thread type $t \in \mathbb{T}$. The empty event $\mathcal{E}$ is a special event that indicates that no events are matched. Then, we assume a total function $\mathrm{ev}_t : \mathbb{A} \to \{\mathcal{E}\} \cup \mathbb{E}_t$. The implementation of ev relies on the specification formalism used, it is capable of generating events based on the context of the action itself. For example, the conversion can utilize the runtime context of actions to generate parametric events when needed. We illustrate a function ev that matches using the label of an action in Ex. 2.

Example 2 (Events.). We identify for *readers-writers* (Ex. 1) two thread types: $\mathbb{T}_{rw} \stackrel{\text{def}}{=} \{\mathrm{reader}, \mathrm{writer}\}$. We are interested in the events $\mathbb{E}_{\mathrm{reader}} \stackrel{\text{def}}{=} \{\mathrm{read}\}$, and $\mathbb{E}_{\mathrm{writer}} \stackrel{\text{def}}{=} \{\mathrm{write}\}$. For a specification at the level of a given thread, we have either a reader or a writer, and the event associated with the reader (resp. writer) is read (resp. write).

$$\mathrm{ev}_{\mathrm{reader}}(\mathrm{a}) \stackrel{\text{def}}{=} \begin{cases} \mathrm{read} & \text{if a.lbl} = \text{“r”}, \\ \mathcal{E} & \text{otherwise} \end{cases} \qquad \mathrm{ev}_{\mathrm{writer}}(\mathrm{a}) \stackrel{\text{def}}{=} \begin{cases} \mathrm{write} & \text{if a.lbl} = \text{“w”}, \\ \mathcal{E} & \text{otherwise.} \end{cases}$$

3.2 Scopes: Properties Over Concurrent Regions

We now define the notion of *scope*. A scope defines a projection of the concurrent execution to delimit concurrent regions and allow verification to be performed at the level of regions instead of the entire execution.

Synchronizing Actions A scope s is associated with a synchronizing predicate $\mathrm{sync}_{\mathrm{s}} : \mathbb{A} \to \mathbb{B}_2$ which is used to determine *synchronizing actions* (SAs). The set of synchronizing actions for a scope s is defined as: $\mathrm{SA}_{\mathrm{s}} = \{a \in \mathbb{A} \mid \mathrm{sync}_{\mathrm{s}}(a) = \top\}$. SAs constitute synchronization points in a concurrent execution for multiple threads. A valid set of SAs is such that there exists a total order on all actions in the set (i.e., no two SAs can occur concurrently). As such SAs are sequenced and can be mapped to indices. Function $\mathrm{idx}_{\mathrm{s}} : \mathrm{SA}_{\mathrm{s}} \to \mathbb{N} \setminus \{0\}$ returns the index of a synchronizing action. For convenience, we map them starting at 1, as 0 will indicate the initial state. We denote by $|\mathrm{idx}_{\mathrm{s}}|$ the length of the sequence.

Scope Region A scope region selects actions of the concurrent execution delimited by two successive SAs. We define two "special" synchronizing actions: $\mathrm{begin}, \mathrm{end} \in \mathbb{A}$ common to all scopes that are needed to evaluate the first and last region. The actions refer to the beginning and end of the concurrent execution, respectively.

Definition 5 (Scope Regions). *Given a scope* s *and an associated index function* $\mathrm{idx_s} : \mathrm{SA_s} \to \mathbb{N}\backslash\{0\}$*, the scope regions are given by function* $\mathcal{R}_\mathrm{s} : \mathrm{codom}(\mathrm{idx_s}) \cup \{0, |\mathrm{idx_s}| + 1\} \to 2^{\mathbb{A}}$*, defined as:*

$$\mathcal{R}_\mathrm{s}(i) \stackrel{\text{def}}{=} \begin{cases} \{a \in \mathbb{A} \mid \langle a', a\rangle \in \to \wedge \langle a, a''\rangle \in \to \wedge\, \mathrm{issync}(a', i-1) \\ \wedge\, \mathrm{issync}(a'', i)\} & \textit{if } 1 \le i \le |\mathrm{idx_s}|, \\ \{a \in \mathbb{A} \mid \langle a', a\rangle \in \to \wedge \langle a, \mathrm{end}\rangle \in \to \wedge\, \mathrm{issync}(a', i-1)\} & \textit{if } i = |\mathrm{idx_s}| + 1, \\ \{a \in \mathbb{A} \mid \langle \mathrm{begin}, a\rangle \in \to \wedge \langle a, a''\rangle \in \to \wedge\, \mathrm{issync}(a'', 1)\} & \textit{if } i = 0, \\ \emptyset & \textit{otherwise} \end{cases}$$

where: $\mathrm{issync}(a, i) \stackrel{\text{def}}{=} (\mathrm{sync_s}(a) = \top \wedge \mathrm{idx_s}(a) = i)$.

$\mathcal{R}_\mathrm{s}(i)$ is the i-th scope region, the set of all actions that happened between the two synchronizing actions a and a', where $\mathrm{idx_s}(a) = i$ and $\mathrm{idx_s}(a') = i+1$ taking into account the start and end of a program execution (i.e., actions begin and end, respectively).

Example 3 (Scope regions). For *readers-writers* (Ex. 1), we consider the resource service lock (s) to be the one of interest, as it delimits the concurrent regions that allow either a writer to write or readers to read. We label the scope by res for the remainder of the paper. The synchronizing predicate $\mathrm{sync_{res}}$ selects all actions with label l (lock acquire) and with the lock id s present in the context of the action. The obtained sequence of SAs is $\mathtt{0.l}^0_\mathtt{s} \cdot \mathtt{1.l}^1_\mathtt{s} \cdot \mathtt{2.l}^0_\mathtt{s}$. The value of $\mathrm{idx_{res}}$ for each of the obtained SAs is respectively 1, 2, and 3. Every lock acquire delimits the regions of the concurrent execution. The region $k+1$ includes all actions between the two lock acquires $\mathtt{0.l}^0_\mathtt{s}$ and $\mathtt{1.l}^1_\mathtt{s}$. That is, $\mathcal{R}_\mathrm{res}(k+1) = \{\mathtt{0.w}^0, \mathtt{0.u}^0_\mathtt{s}, \mathtt{0.u}^0_\mathtt{t}, \mathtt{1.l}^1_\mathtt{t}, \mathtt{0.l}^1_\mathtt{c}, \mathtt{0.i}^1\}$. The region $k+2$ contains two concurrent reads: $\mathtt{r}^1, \mathtt{r}^2$.

Definition 6 (Scope fragment). *The scope fragment associated with a scope region* $\mathcal{R}_\mathrm{s}(i)$ *is defined as* $\mathcal{F}_\mathrm{s}(i) \stackrel{\text{def}}{=} \langle \mathcal{R}_\mathrm{s}(i), \to \cap\, \mathcal{R}_\mathrm{s}(i) \times \mathcal{R}_\mathrm{s}(i)\rangle$.

Proposition 1 (Scope fragment preserves order). *Given a scope* s*, we have:*
$\forall i \in \mathrm{dom}(\mathcal{R}_\mathrm{s}(i)) : \mathrm{snd}(\langle \mathbb{A}, \to\rangle, \mathcal{F}_\mathrm{s}(i)) \wedge \mathrm{faith}(\langle \mathbb{A}, \to\rangle, \mathcal{F}_\mathrm{s}(i))$.

Proposition 1 states that for a given scope, any fragment (obtained using $\mathcal{F}_\mathrm{s}$) is a sound and faithful trace of the concurrent execution. This is ensured by construction using Definitions 5 and 6 which follow the same principles of the definitions of soundness (Definition 3) and faithfulness (Definition 4).

Remark 1. In this paper, scopes regions are defined by the user by selecting the synchronizing predicate as part of the specification. Given a property, regions should delimit events whose order is important for a property. For instance, for a property specifying that *"between each write, at least one read should occur"*, the scope regions should delimit read versus write events. Delimiting the read events themselves, performed by

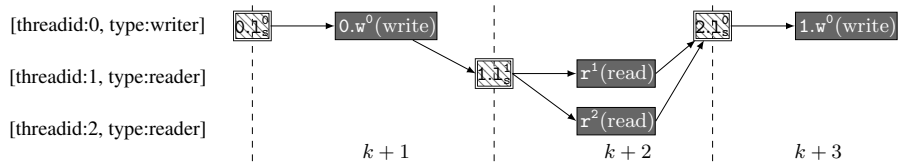

Fig. 3: Projected actions using the scope and local properties of 1-Writer 2-Readers. The action labels $\mathrm{l}, \mathrm{w}, \mathrm{r}$ indicate respectively the following: lock, write, and read. Filled actions indicate actions for which function ev for the thread type returns an event. Actions with a pattern background indicate the SAs for the scope.

different threads, is not significant. How to analyze the program to find and suggest scopes for the user that are suitable for monitoring a given property is an interesting challenge that we leave for future work. Moreover, we assume the program is properly synchronized and free from data races.

Local Properties In a given scope region, we determine properties that will be checked locally on each thread. A thread-local monitor checks a local property independently for each given thread. These properties can be seen as the analogous of *per-thread* monitoring applied between two SAs. For a specific thread, we have a guaranteed total order on the local actions being formed. This ensures that local properties are compatible and can be checked with existing RV techniques and formalisms. We refer to those properties as *local properties*.

Definition 7 (Local property). *A local property is a tuple* $\langle \mathrm{type}, \mathrm{EVS}, \mathrm{RT}, \mathrm{eval} \rangle$ *with:*

- $\mathrm{type} \in \mathbb{T}$ *is the thread type to which the local property applies;*
- $\mathrm{EVS} \subseteq \mathbb{E}_{\mathrm{type}}$ *is a subset of (thread type) events relevant to the property evaluation;*
- RT *is the resulting type of the evaluation (called* return type*); and*
- $\mathrm{eval} : (\mathbb{N} \to \mathrm{EVS}) \to \mathrm{RT}$ *is the evaluation function of the property, taking as input a sequence of events, and returning the result of the evaluation.*

We use the dot notation: for a given property $\mathrm{prop} = \langle \mathrm{type}, \mathrm{EVS}, \mathrm{RT}, \mathrm{eval} \rangle$ we use $\mathrm{prop.type}$, $\mathrm{prop.EVS}$, $\mathrm{prop.RT}$, and $\mathrm{prop.eval}$ respectively.

Example 4 (At least one read). The property "at least one read", defined for the thread type reader, states that a reader must perform at least one read event. It can be expressed using the classical LTL_3 [10] (a variant of linear temporal logic with finite-trace semantics commonly used in RV) as $\varphi_{1\mathrm{r}} \stackrel{\mathrm{def}}{=} \mathbf{F}(\mathrm{read})$ using the set of atomic propositions $\{\mathrm{read}\}$. Let $\mathrm{LTL}_3{}^{\mathrm{AP}}_{\varphi}$ denote the evaluation function of LTL_3 using the set of atomic propositions AP and a formula φ, and let $\mathbb{B}_3 = \{\top, \bot, ?\}$ be the truth domain where $?$ denotes an inconclusive verdict. To check on readers, we specify it as the local property: $\langle \mathrm{reader}, \{\mathrm{read}\}, \mathbb{B}_3, \mathrm{LTL}_3{}^{\{\mathrm{read}\}}_{\varphi_{1\mathrm{r}}} \rangle$. Similarly, we can define the local specification for at least one write.

Scope Trace To evaluate a local property, we restrict the trace to the actions of a given thread contained within a scope region. A scope trace is analogous to acquiring the trace for *per-thread* monitoring [5, 30] in a given scope region (see Definition 5). The scope trace is defined as a projection of the concurrent execution, on a specific thread, selecting actions that fall between two synchronizing actions.

Definition 8 (Scope trace). *Given a local property* $\mathrm{p} = \langle \mathrm{type}, \mathrm{EVS}, \mathrm{RT}, \mathrm{eval} \rangle$ *in a scope region* $\mathcal{R}_\mathrm{s}$ *with index* i*, a* scope trace *is obtained using the projection function* proj*, which outputs the sequence of actions of length* n *for a given thread with* $\mathrm{tid} \in \mathrm{TID}$ *that are associated with events for the property. We have:* $\forall \ell \in [0, n]$

$$\mathrm{proj}(\mathrm{tid}, i, \mathrm{p}, \mathcal{R}_\mathrm{s}) \stackrel{\text{def}}{=} \begin{cases} \mathrm{filter}(a_0) \cdot \ldots \cdot \mathrm{filter}(a_n) & \textit{if } i \in \mathrm{dom}(\mathcal{R}_\mathrm{s}) \wedge \mathrm{tid} \in \mathrm{pool}(\mathrm{type}), \\ \mathcal{E} & \textit{otherwise}, \end{cases}$$

$$\textit{with: } \mathrm{filter}(a_\ell) \stackrel{\text{def}}{=} \begin{cases} e & \textit{if } \mathrm{ev}_\mathrm{type}(a_\ell) \in \mathrm{EVS} \\ \mathcal{E} & \textit{otherwise}, \end{cases}$$

where $\cdot$ *is the sequence concatenation operator (such that* $a \cdot \mathcal{E} = \mathcal{E} \cdot a = a$*), with* $(\forall j \in [1, n] : \langle a_{j-1}, a_j \rangle \in \rightarrow) \wedge (\forall k \in [0, n] : a_k \in \mathcal{R}_\mathrm{s}(i) \wedge a_k.\mathrm{threadid} = \mathrm{tid})$.

For a given thread, the scope trace filters the actions associated with an event for the local property (i.e., $\mathrm{ev}_\mathrm{type}(a_\ell) \in \mathrm{EVS}$) of a scope region. It includes only actions that are associated with the *threadid* that has the correct type associated with the local specification (i.e., $\mathrm{tid} \in \mathrm{pool}(\mathrm{type})$). While the scope trace is obtained using projection, it is still needed to convert actions to events to later evaluate local properties, to do so we generate the sequence of events associated with the actions in the projected trace. That is, for a given action a in the sequence, we output $\mathrm{ev}_\mathrm{type}(a_\ell)$, we denote the generated sequence as $\mathrm{evs}(\mathrm{proj}(\mathrm{tid}, i, \mathrm{p}, \mathcal{R}_\mathrm{s}))$.

Example 5 (Scope trace). Figure 3 illustrates the projection on the scope regions defined using the resource lock (Ex. 3) for each of the 1 writer and 2 reader threads, where the properties "at least one write" or "at least one read" (Example 4) apply. We see the scope traces for region $k + 1$ are respectively $\mathtt{0.w}^0, \mathcal{E}, \mathcal{E}$ for the threads with thread ids 0, 1, and 2 respectively. For that region, we can now evaluate the local specification independently for each thread on the resulting traces by converting the sequences of events: $\mathrm{write}, \mathcal{E}, \mathcal{E}$ for each of the scope traces.

Proposition 2 (proj **preserves per-thread order).** *Given a scope* s*, a thread with threadid* tid*, and a local property* p*, we have:*
$\forall i \in \mathrm{dom}(\mathcal{R}_\mathrm{s}) : \mathrm{snd}\left(\langle \mathbb{A}, \rightarrow \rangle, \mathrm{proj}(\mathrm{tid}, i, \mathrm{p}, \mathcal{R}_\mathrm{s})\right) \wedge \mathrm{faith}\left(\langle \mathbb{A}, \rightarrow \rangle, \mathrm{proj}(\mathrm{tid}, i, \mathrm{p}, \mathcal{R}_\mathrm{s})\right)$.

Proposition 2 is guaranteed by construction (from Definition 8), ensuring that projection function proj does not produce any new actions and does not change any order information from the point of view of a given thread. We also note the assumption that for a single thread, all its actions are totally ordered, and therefore we capture all possible order information for the actions in the scope region. Finally, the function filter only suppresses actions that are not relevant to the property, without adding or re-ordering actions. The sequence of events obtained using the function evs also follows the same order.

Scope State A scope state aggregates the result of evaluating all local properties for a given scope region. To define a scope state, we consider a scope s, with a list of local properties $\langle \mathrm{prop}_0, \ldots, \mathrm{prop}_n \rangle$ of return types respectively $\langle \mathrm{RT}_0, \ldots, \mathrm{RT}_n \rangle$. Since a local specification can apply to an arbitrary number of threads during the execution, for each specification we create the type as a dictionary binding a *threadid* to the return type (represented as a total function). We use the type na to determine a special type indicating the property does not apply to the thread (as the thread type does not match the property). We can now define the return type of evaluating all local properties as $\mathrm{RI} \stackrel{\text{def}}{=} \langle \mathrm{TID} \to \{\mathrm{na}\} \cup \mathrm{RT}_0, \ldots, \mathrm{TID} \to \{\mathrm{na}\} \cup \mathrm{RT}_n \rangle$. Function $\mathrm{state_s} : \mathrm{RI} \to \mathbb{I}_\mathrm{s}$ processes the result of evaluating local properties to create a scope state in $\mathbb{I}_\mathrm{s}$.

Example 6 (Scope state). We illustrate the scope state by evaluating the properties "at least one read" (p_r) and "at least one write" (p_w) (Ex. 4) on scope region $k+2$ in Fig. 3. We have $\mathrm{TID} = \{0, 1, 2\}$, we determine for each reader the trace (being (read) for both), and the writer being empty (i.e. no write was observed). As such for property p_r (resp. p_w), we have the result of the evaluation $[0 \mapsto \mathrm{na}, 1 \mapsto \top, 2 \mapsto \top]$ (resp. $[0 \mapsto ?, 1 \mapsto \mathrm{na}, 2 \mapsto \mathrm{na}]$). We notice that for property p_r, the thread of type writer evaluates to na, as it is not concerned with the property.

We now consider the state creation function $\mathrm{state_s}$. We consider the following atomic propositions $\mathrm{activereader}$, $\mathrm{activewriter}$, $\mathrm{allreaders}$, and $\mathrm{onewriter}$ that indicate respectively: at least one thread of type reader performed a read, at least one thread of type writer performed a write, all threads of type reader ($|\mathrm{pool}(\mathrm{reader})|$) performed at least a read, and at most one thread of type writer performed a write. The scope state in this case is a list of 4 boolean values indicating each atomic proposition respectively. As such by counting the number of threads associated with $\top$, we can compute the Boolean value of each atomic proposition. For region $k+2$, we have the following state: $\langle \top, \bot, \top, \bot \rangle$. We can establish a total order of scope states. For $k+1$, $k+2$ and $k+3$, we have the sequence $\langle \bot, \top, \bot, \top \rangle \cdot \langle \top, \bot, \top, \bot \rangle \cdot \langle \bot, \top, \bot, \top \rangle$.

We are now able to define formally a scope by associating an identifier with a synchronizing predicate, a list of local properties, a spawn predicate, and a scope property evaluation function. We denote by SID the set of scope identifiers.

Definition 9 (Scope). *A scope is a tuple* $\langle \mathrm{sid}, \mathrm{sync_{sid}}, \langle \mathrm{prop}_1, \ldots, \mathrm{prop}_n \rangle, \mathrm{state_{sid}}, \mathrm{seval_{sid}} \rangle$*, where:*

- $\mathrm{sid} \in \mathrm{SID}$ *is the scope identifier;*
- $\mathrm{sync_{sid}} : \mathbb{A} \to \mathbb{B}_2$ *is the synchronizing predicate that determines SAs;*
- $\langle \mathrm{prop}_0, \ldots, \mathrm{prop}_n \rangle$ *is a list of local properties (Definition 7);*
- $\mathrm{state_{sid}} : \langle \mathrm{TID} \to \{\mathrm{na}\} \cup \mathrm{prop}_0.\mathrm{RT}, \ldots, \mathrm{TID} \to \{\mathrm{na}\} \cup \mathrm{prop}_n.\mathrm{RT} \rangle \to \mathbb{I}_\mathrm{s}$ *is the scope state creation function;*
- $\mathrm{seval_{sid}} : \mathbb{N} \times \mathbb{I}_\mathrm{s} \to \mathbb{O}$ *is the evaluation function of the scope property over a sequence of scope states.*

3.3 Semantics for Evaluating Scopes

After defining scope states, we are now able to evaluate properties on the scope. To evaluate a scope property, we first evaluate each local property for the scope region, we

then use $\mathrm{state}_{\mathrm{sid}}$ to generate the scope state for the region. After producing the sequence of scope states, the function $\mathrm{seval}_{\mathrm{sid}}$ evaluates the property at the level of a scope.

Definition 10 (Evaluating a scope property). *Using the synchronizing predicate* $\mathrm{sync}_{\mathrm{sid}}$*, we obtain the regions* $\mathcal{R}_{\mathrm{sid}}(i)$ *for* $i \in [0, m]$ *with* $m = |\mathrm{idx}_{\mathrm{sid}}| + 1$*. The evaluation of a scope property (noted* res*) for the scope* $\langle \mathrm{sid}, \mathrm{sync}_{\mathrm{sid}}, \langle \mathrm{prop}_0, \ldots, \mathrm{prop}_n \rangle, \mathrm{state}_{\mathrm{sid}}, \mathrm{seval}_{\mathrm{sid}} \rangle$ *is computed as:* $\forall tid \in \mathrm{TID}, \forall j \in [0, n]$

$$\mathrm{res} = \mathrm{seval}_{\mathrm{sid}}(\mathrm{SR}_0 \cdot \ldots \cdot \mathrm{SR}_m), \text{ where } \mathrm{SR}_i = \mathrm{state}_{\mathrm{sid}}(\langle \mathrm{LR}_0^i, \ldots, \mathrm{LR}_n^i \rangle)$$

$$\mathrm{LR}_j^i = \begin{cases} tid \mapsto \mathrm{prop}_j.\mathrm{eval}(\mathrm{evs}(\mathrm{proj}(tid, i, \mathrm{prop}_j, \mathcal{R}_{\mathrm{sid}}))) & \text{if } tid \in \mathrm{pool}(\mathrm{prop}_j.\mathrm{type}) \\ tid \mapsto \mathrm{na} & \text{otherwise} \end{cases}$$

Example 7 (Evaluating scope properties). We use LTL to formalize three scope properties based on the scope states from Ex. 6 operating on the alphabet $\{\mathrm{activereader}, \mathrm{activewriter}, \mathrm{allreaders}, \mathrm{onewriter}\}$:

- Mutual exclusion between readers and writers: $\varphi_0 \stackrel{\mathrm{def}}{=} \mathrm{activewriter}\ \mathbf{XOR}\ \mathrm{activereader}$.
- Mutual exclusion between writers: $\varphi_1 \stackrel{\mathrm{def}}{=} \mathrm{activewriter} \implies \mathrm{onewriter}$.
- All readers must read a written value: $\varphi_2 \stackrel{\mathrm{def}}{=} \mathrm{activereader} \implies \mathrm{allreaders}$.

Therefore the specification is: $\mathbf{G}(\varphi_0 \wedge \varphi_1 \wedge \varphi_2)$. We recall that a scope state is a list of boolean values for the atomic propositions in the following order: $\mathrm{activereader}$, $\mathrm{activewriter}$, $\mathrm{allreaders}$, and $\mathrm{onewriter}$. The sequence of scope states from Ex. 6: $\langle \bot, \top, \bot, \top \rangle \cdot \langle \top, \bot, \top, \bot \rangle \cdot \langle \bot, \top, \bot, \top \rangle$ complies with the specification.

Correctness of Scope Evaluation We assume that the SAs selected by the user in the specification are totally ordered. This ensures that the order of the scope states is a total order, it is then by assumption sound and faithful to the order of the SAs. However, it is important to ensure that the actions needed to construct the state are captured faithfully and in a sound manner. We capture the partial order as follows: (1) actions of different threads are captured in a sound and faithful manner between two successive SAs (Proposition 1), and (2) actions of the same thread are captured in a sound and faithful manner for that thread (Proposition 2). Furthermore, we are guaranteed by Definition 10 that each local property evaluation function is passed to all actions relevant to the given thread (and no other). As such, for the granularity level of the SAs, we obtain all relevant order information.

Evaluating without resetting. We notice that in Definition 10 monitors on local properties are reset for each concurrency region. As such, they are unable to express properties that span multiple concurrency regions of the same thread. The semantics of function res conceptually focus on treating concurrency regions independently. However, we can account for elaborating the expressiveness of local properties by extending the alphabet for each local property with the atomic proposition sync which delimits the concurrency region. The proposition sync denotes that the scope synchronizing action has occurred, and adds it to the trace. We need to take careful consideration that threads may sleep and not receive any events during a concurrent region. For example, consider two threads waiting on a lock, when one thread gets the lock, the other will not. As such, to pass the sync event to the local specification of the sleeping thread requires we instrument very

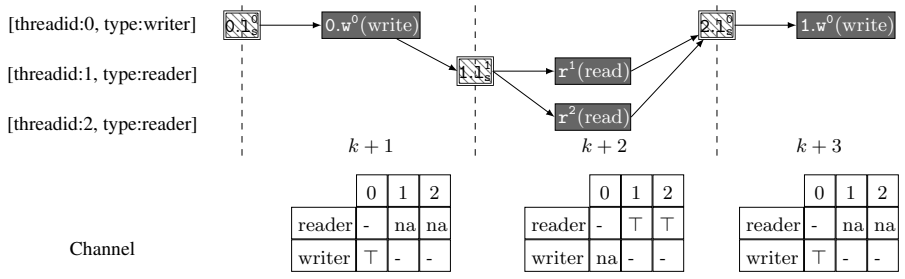

Fig. 4: Example of a scope channel for 1-Writer 2-Readers.

intrusively to account for that, a requirement we do not want to impose. Therefore, we add the restriction that local properties are only evaluated if at least one event relevant to the local property is encountered in the concurrency region (that is not the synchronization event). Using that consideration, we can define an evaluation that considers all events starting from concurrent region 0 up to i, and adding sync events between scopes (we omit the definition for brevity). This allows local monitors to account for synchronization, either to reset or check more expressive specifications such as "*a reader can read at most n times every m concurrency regions*", and "*writers must always write a value that is greater than the last write*".

3.4 Communicating Verdicts and Monitoring

We now proceed to describe how the monitors communicate their verdicts.

Scope channel. The *scope channel* stores information about the scope states during the execution. We associate each scope with a scope channel that has its own timestamp. The channel provides each thread-local monitor with an exclusive memory slot to write its result when evaluating local properties. Each thread can only write to its associated slot in the channel. The timestamp of the channel is readable by all threads participating in the scope but is only incremented by the scope monitor, as we will see.

Example 8 (Scope channel). Figure 4 displays the channel associated with the scope monitoring discussed in Ex. 6. For each scope region, the channel allows each monitor an exclusive memory slot to write its result (if the thread is not sleeping). The slots marked with a dash (-) indicate the absence of monitors. Furthermore, na indicates that the thread was given a slot, but it did not write anything in it (see Definition 10).

For a timestamp t, local monitors no longer write any information for any scope state with a timestamp inferior to t, this makes such states always consistent to be read by any monitor associated with the scope. While this is not in the scope of the paper, it allows monitors to effectively access past data of other monitors consistently.

Thread-local monitors. Each thread-local monitor is responsible for monitoring a local property for a given thread. Recall that each thread is associated with an identifier and a

type. Multiple such monitors can exist on a given thread, depending on the needed properties to check. These monitors are spawned on the creation of the thread. It receives an event, performs checking, and can write its result in its associated scope channel at the current timestamp.

Scope monitors. Scope monitors are responsible for checking the property at the level of the scope. Upon reaching a synchronizing action by any of the threads associated with the scope, the given thread will invoke the scope monitor. The scope monitor relies on the scope channel (shared among all threads) to have access to all observations. Additional memory can be allocated for its own state, but it has to be shared among all threads associated with the scope. The scope monitor is invoked atomically after reaching the scope synchronizing action. First, it constructs the scope state based on the results of the thread-local monitors stored in the scope channel. Second, it invokes the verification procedure on the generated state. Finally, before completing, it increments the timestamp associated with the scope channel.

4 Preliminary Assessment of Overhead

We first opportunistically monitor *readers-writers*, using the specification found in Ex. 7. We then demonstrate our approach with classical concurrent programs[2].

4.1 Readers-Writers

Experiment setup. For this experiment, we utilize the standard LTL_3 semantics defined over the $\mathbb{B}_3$ verdict domain. As such, all the local and scope property types are $\mathbb{B}_3$. We instrument *readers-writers* to insert our monitors and compare our approach to global monitoring using a custom aspect written in AspectJ. In total, we have three scenarios: non-monitored, global, and opportunistic. In the first scenario (non-monitored), we do not perform monitoring. In the second and third scenarios, we perform global and opportunistic monitoring. We recall that global monitoring introduces additional locks at the level of the monitor for all events that occur concurrently. We make sure that the program is well synchronized and data race free with RVPredict [37].

Measures. To evaluate the overhead of our approach, we are interested in defining parameters to characterize concurrency regions found in *readers-writers*. We identify two parameters: the *number of readers* ($\mathrm{nreaders}$), and the *width of the concurrency region* (cwidth). On the one hand, $\mathrm{nreaders}$ determines the maximum parallel threads that are verifying local properties in a given concurrency region. On the other hand, cwidth determines the number of reads each reader performs concurrently when acquiring the lock. Parameter cwidth is measured in number of read events generated. By increasing the size of the concurrency regions, we increase lock contention when multiple concurrent events cause a global monitor to lock. We use a number of writers equivalent to $\mathrm{nreaders} \in \{1, 3, 7, 15, 23, 31, 63, 127\}$ and $\mathrm{cwidth} \in \{1, 5, 10, 15, 30, 60, 100, 150\}$.

[2] The artifact for this paper is available [57].

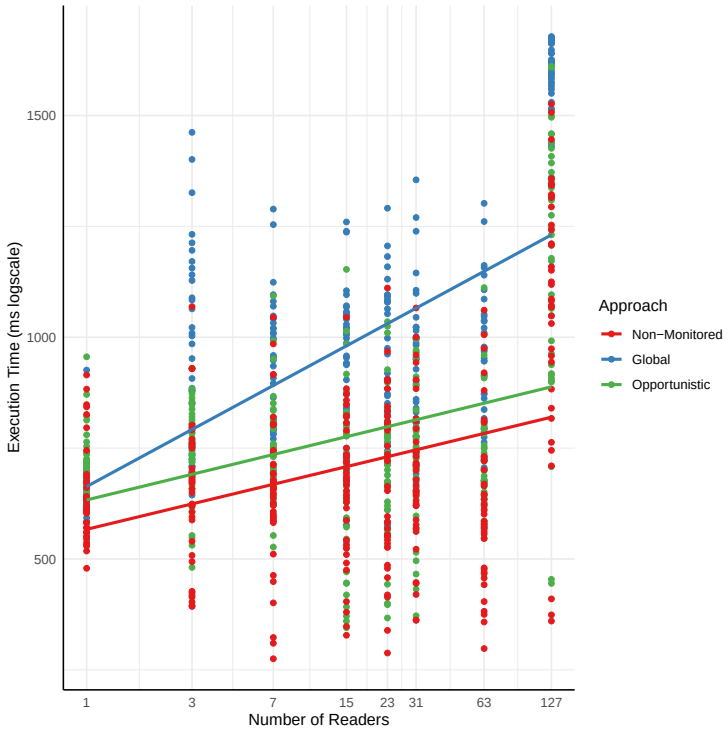

Fig. 5: Execution time for *readers-writers* for non-monitored, global, and opportunistic monitoring when varying the number of readers.

We perform a total of 100,000 writes and 400,000 reads, where reads are distributed evenly across readers. We measure the execution time (in ms) of 50 runs of the program for each of the parameters and scenarios.

Preliminary results. We report the results using the averages while providing the scatter plots with linear regression curves in Figures 5, and 6. Figure 5 shows the overhead when varying the number of readers (nreaders). We notice that for the base program (non-monitored), the execution time increases as lock contention overhead becomes more prominent and the JVM is managing more threads. In the case of global monitoring, as expected we notice an increasing overhead with the increase in the number of threads. As more readers are executing, the program is being blocked on each read which is supposed to be concurrent. For opportunistic, we notice a stable runtime in comparison to the original program as no additional locks are being used; only the delay to evaluate the local and scope properties. Figure 6 shows the overhead when varying the width of the concurrency region (cwidth). We observe that for the base program, the execution time decreases as more reads can be performed concurrently without contention on the shared resource lock. In the case of global monitoring, we also notice a slight decrease, while for opportunistic monitoring, we see a much greater decrease. By increasing the number of concurrent events in a concurrency region, we

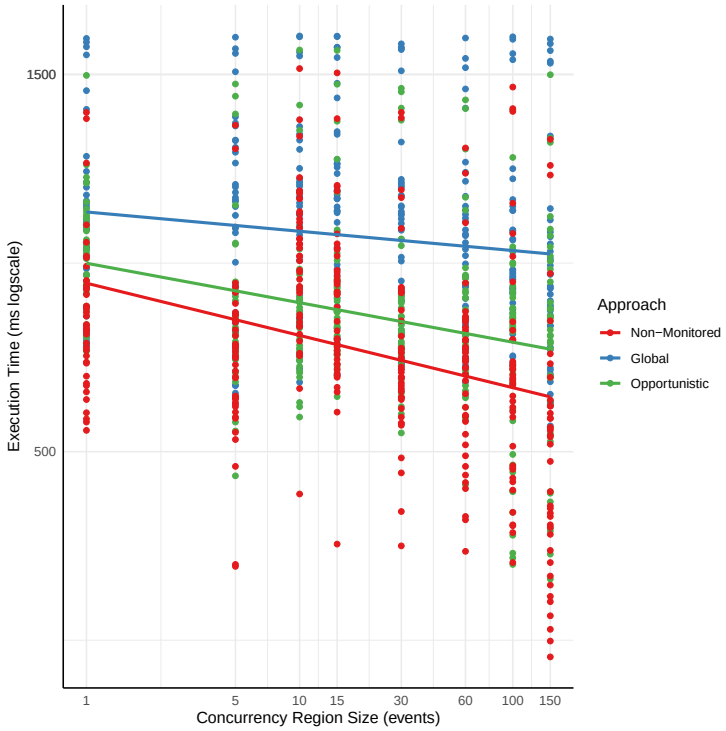

Fig. 6: Execution time varying the number of events in the concurrency region.

highlight the overhead introduced by locking the global monitor. We recall that a global monitor must lock to linearize the trace, and as such interferes with concurrency. This can be seen by looking at the two curves for global and opportunistic monitoring, we see that opportunistic closely follows the speedup of the non-monitored program, while global monitoring is much slower. For opportunistic monitoring, we expected a positive performance payoff when events in concurrency regions are dense.

4.2 Other Benchmarks

We target classical benchmarks that use different concurrency primitives to synchronize threads. We perform global and opportunistic monitoring and report our results using the averages of 100 runs in Figure 7. We use an implementation of the Bakery lock algorithm [40], for two threads *2-bakery* and n threads *n-bakery*. The algorithm performs synchronization using reads and writes on shared variables and guarantees mutual exclusion on the critical section. As such, we monitor the program for the *bounded waiting* property which specifies that a process should not wait for more than a limited number of turns before entering the critical section. For opportunistic monitoring, thread-local monitors are deployed on each thread to monitor if the thread acquires the critical section. Scope monitors check if a thread is waiting for more than n turns before entering the critical section. We notice slightly less overhead with opportunistic than global for

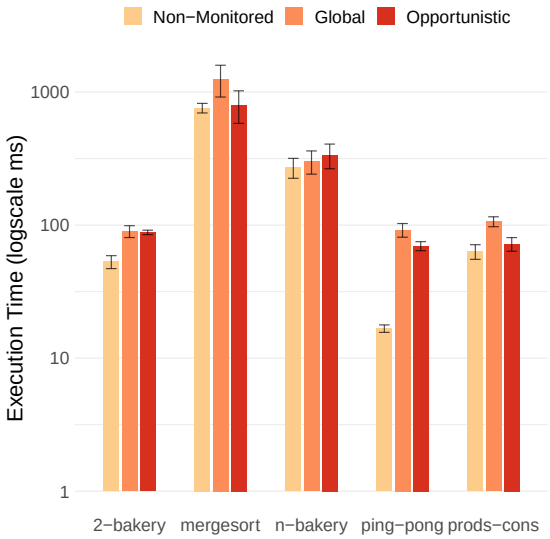

Fig. 7: Execution time of benchmarks.

2-bakery and more overhead with opportunistic on *n-bakery*. This is because of the small concurrency region (cwidth) which is equal to 1. As such, the overhead of evaluating local and scope monitors by several threads, having a cwidth of 1, exceeds the gain in performance achieved by our approach and hence not fitting for opportunistic monitoring.

We also monitor a textbook example of Ping-Pong algorithm [33] that is used for instance in databases and routing protocols. The algorithm synchronizes, using reads and writes on shared variables and busy waiting, between two threads producing events pi for the pinging thread and po for the pong thread. We monitor for the *alternation* property specified as $\varphi \stackrel{\text{def}}{=} (\mathrm{ping} \implies \mathbf{X}\mathrm{pong}) \wedge (\mathrm{pong} \implies \mathbf{X}\mathrm{ping})$. We also include a classic producer-consumer program from [35] which uses a concurrent FIFO queue using locks and conditions. We monitor the *precedence* property, which specifies the requirement that a consume (event c) is preceded by a produce (event p), expressed in LTL as $\neg c\, \mathbf{W}\, p$. For both above benchmarks, we observe less overhead when monitoring with opportunistic, since no additional locks are being enforced on the execution.

We also monitor a parallel mergesort algorithm which is a divide-and-conquer algorithm to sort an array. The algorithm uses the fork-join framework [42] which recursively splits the array into sorting tasks that are handled by different threads. We are interested in monitoring if a forked task is returning a correctly sorted array before performing a merge. The monitoring step is expensive and linear in the size of the array as it involves scanning it. For opportunistic, we use the joining of two subtasks as our synchronizing action and deploy scope monitors at all levels of the recursive hierarchy. We observe less overhead when monitoring with opportunistic than global monitoring, as concurrent threads do not have to wait at each monitoring step. This

benchmark motivates us to further investigate other hierarchical models of computation where opportunistic RV can be used such as [22].

5 Related Work

We focus here on techniques developed for the verification of behavioral properties of multithreaded programs written in Java and refer to [12] for a detailed survey on tools covering generic concurrency errors. The techniques we cover typically analyze a trace to either *detect* or *predict* violations.

Java-MOP [18], Tracematches [5,13], MarQ [52], and LARVA [21] chosen from the RV competitions [8,26,53] are runtime monitoring tools for violation detection. These tools allow different specification formalisms such as finite-state machines, extended regular expressions, context-free grammars, past-time linear temporal logic, and Quantified Event Automata (QEA) [6]. Their specifications rely on a total order of events and require that a collected trace be linearized. They were initially developed to monitor single-threaded programs and later adapted to monitor multithreaded programs. As mentioned, to monitor global properties spanning multiple threads these techniques impose a lock on each event blocking concurrent regions in the program and forcing threads to synchronize. Moreover, they often produce inconsistent verdicts with the existence of concurrent events [23]. EnforceMOP [45] for instance, can be used to detect and enforce properties (deadlocks as well). It controls the runtime scheduler and blocks the execution of threads that might cause a property violation, sometimes itself leading to a deadlock.

Predictive techniques [19, 31, 39, 55] reason about all feasible interleavings from a recorded trace of a single execution. As such, they need to establish the causal ordering between the actions of the program. These tools implement vector clock algorithms, such as [54], to timestamp events. The algorithm blocks the execution on each property event and also on all synchronizing actions such as reads and writes. Vector clock algorithms typically require synchronization between the instrumentation, program actions, and algorithm's processing to avoid data races [16]. jPredictor [19] for instance, uses sliced causality [17] to prune the partial order such that only relevant synchronization actions are kept. This is achieved with the help of static analysis and after recording at least one execution of the program. The tool is demonstrated on atomicity violations and data races; however, we are not aware of an application in the context of generic behavioral properties. RVPredict [37] develops a sound and maximal causal model to analyze concurrency in a multithreaded program. The correct behavior of a program is modeled as a set of logical constraints, thus restricting the possible traces to consider. Traces are ordered permutations containing both control flow operations and memory accesses and are constrained by axioms tailored to data race and sequential consistency. The theory supports any logical constraints to determine correctness, it is then possible to encode a specification on multithreaded programs as such. However, allowing for arbitrary specifications to be encoded while supported in the model, is not supported in the provided tool (RVPredict). In [27], the authors present ExceptioNULL that target null-pointer exceptions. Violations and causality are represented as constraints over actions, and the feasibility of violations is explored via an SMT constraint solver. GPredict [36]

extends the specification formalism past data races to target generic concurrency properties. GPredict presents a generic approach to reason about behavioral properties and hence constitutes a monitoring solution when concurrency is present. Notably, GPredict requires specifying thread identifiers explicitly in the specification. This makes specifications with multiple threads to become extremely verbose; unable to handle a dynamic number of threads. For example, in the case of *readers-writers*, adding extra readers or writers requires rewriting the specification and combining events to specify each new thread. The approach behind GPredict can also be extended to become more expressive, e.g. to support counting events to account for fairness in a concurrent setting. Furthermore, GPredict relies on recording a trace of a program before performing an offline analysis to determine concurrency errors [36]. In addition to being incomplete due to the possibility of not getting results from the constraint solver, the analysis from GPredict might also miss some order relations between events resulting in false positives. In general, the presented predictive tools are often designed to be used offline and unfortunately, many of them are no longer maintained.

In [14,15], the authors present monitoring for *hyperproperties* written in alternation-free fragments of HyperLTL [20]. Hyperproperties are specified over sets of execution traces instead of a single trace. In our setup, each thread is producing its trace and thus scope properties we monitor can be expressed in HyperLTL for instance. The time occurrence of events will be delimited by concurrency regions and thus traces will consist of propositions that summarize the concurrency region. We have yet to explore the applicability of specifying and monitoring hyperproperties within our opportunistic approach.

6 Conclusion and Perspectives

We introduced a generic approach for the online monitoring of multithreaded programs. Our approach distinguishes between thread-local properties and properties that span concurrency regions referred to as scopes (both types of properties can be monitored with existing tools). Our approach relies heavily on existing totally ordered operations in the program. However, by utilizing the existing synchronization, we can monitor online while leveraging both existing per-thread and global monitoring techniques. Finally, our preliminary evaluation suggests that opportunistic monitoring incurs a lower overhead in general than classical monitoring.

While the preliminary results are promising, additional work needs to be invested to complete the automatic synthesis and instrumentation of monitors. So far, splitting the property over local and scope monitors is achieved manually and scope regions are guaranteed by the user to follow a total order. Analyzing the program to find and suggest scopes suitable for splitting and monitoring a given property is an interesting challenge that we leave for future work. The program can be run, for instance, to capture its causality and recommend suitable synchronization actions for delimiting scope regions. Furthermore, the expressiveness of the specification can be increased by extending scopes to contain other scopes and adding more levels of monitors. This allows for properties that target not just thread-local properties, but also concurrent regions enclosed in other concurrent regions, thus creating a hierarchical setting.

References

1. Patterns in property specifications for finite-state verification home page. https://matthewbdwyer.github.io/psp/patterns.html, https://matthewbdwyer.github.io/psp/patterns.html
2. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: a tutorial. Computer 29(12), 66–76 (Dec 1996)
3. Agarwal, A., Garg, V.K.: Efficient dependency tracking for relevant events in shared-memory systems. In: Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing. p. 19–28. PODC '05, Association for Computing Machinery, New York, NY, USA (2005), https://doi.org/10.1145/1073814.1073818
4. 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)
5. 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)
6. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.E.: Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In: 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, pp. 68–84. Springer (2012), https://doi.org/10.1007/978-3-642-32759-9_9
7. 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)
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, E., Falcone, Y. (eds.) Lectures on Runtime Verification - Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457, pp. 1–33. Springer (2018)
10. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for ltl and tltl. ACM Trans. Softw. Eng. Methodol. 20(4), 14:1–14:64 (Sep 2011)
11. Bensalem, S., Havelund, K.: Dynamic deadlock analysis of multi-threaded programs. In: Proceedings of the First Haifa International Conference on Hardware and Software Verification and Testing. p. 208–223. HVC'05, Springer-Verlag, Berlin, Heidelberg (2005), https://doi.org/10.1007/11678779_15
12. Bianchi, F.A., Margara, A., Pezzè, M.: A survey of recent trends in testing concurrent software systems. IEEE Transactions on Software Engineering 44(8), 747–783 (2018)
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. Bonakdarpour, B., Sanchez, C., Schneider, G.: Monitoring hyperproperties by combining static analysis and runtime verification. In: Leveraging Applications of Formal Methods, Verification and Validation. Verification: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II. p. 8–27. Springer-Verlag, Berlin, Heidelberg (2018), https://doi.org/10.1007/978-3-030-03421-4_2

15. Brett, N., Siddique, U., Bonakdarpour, B.: Rewriting-based runtime verification for alternation-free hyperltl. In: Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10206. p. 77–93. Springer-Verlag, Berlin, Heidelberg (2017), https://doi.org/10.1007/978-3-662-54580-5_5
16. Cain, H.W., Lipasti, M.H.: Verifying sequential consistency using vector clocks. In: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures. p. 153–154. SPAA '02, Association for Computing Machinery, New York, NY, USA (2002), https://doi.org/10.1145/564870.564897
17. Chen, F., Roşu, G.: Parametric and sliced causality. In: Proceedings of the 19th International Conference on Computer Aided Verification. p. 240–253. CAV'07, Springer-Verlag, Berlin, Heidelberg (2007)
18. 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)
19. Chen, F., Serbanuta, T.F., Rosu, G.: Jpredictor: A predictive runtime analysis tool for java. In: Proceedings of the 30th International Conference on Software Engineering. p. 221–230. ICSE '08, Association for Computing Machinery, New York, NY, USA (2008), https://doi.org/10.1145/1368088.1368119
20. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. 18(6), 1157–1210 (sep 2010)
21. 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
22. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (jan 2008), https://doi.org/10.1145/1327452.1327492
23. El-Hokayem, A., Falcone, Y.: Can we monitor all multithreaded programs? In: Colombo, C., Leucker, M. (eds.) Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11237, pp. 64–89. Springer (2018), https://doi.org/10.1007/978-3-030-03769-7_6
24. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Broy, M., Peled, D.A., Kalus, G. (eds.) Engineering Dependable Software Systems, NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 34, pp. 141–175. IOS Press (2013)
25. Falcone, Y., Krstic, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. In: Colombo, C., Leucker, M. (eds.) Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings. Lecture Notes in Computer Science, vol. 23, pp. 241–262. Springer (2018)
26. 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)
27. Farzan, A., Parthasarathy, M., Razavi, N., Sorrentino, F.: Predicting null-pointer dereferences in concurrent programs. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. FSE '12, Association for Computing Machinery, New York, NY, USA (11 2012), https://doi.org/10.1145/2393596.2393651
28. Flanagan, C., Freund, S.N.: Atomizer: A dynamic atomicity checker for multithreaded programs. SIGPLAN Not. 39(1), 256–267 (jan 2004), https://doi.org/10.1145/982962.964023

29. Flanagan, C., Freund, S.N.: Fasttrack: Efficient and precise dynamic race detection. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 121–133. PLDI '09, Association for Computing Machinery, New York, NY, USA (2009), https://doi.org/10.1145/1542476.1542490
30. Formal Systems Laboratory: JavaMOP4 Syntax (2018), http://fsl.cs.illinois.edu/index.php/JavaMOP4_Syntax
31. Gao, Q., Zhang, W., Chen, Z., Zheng, M., Qin, F.: 2ndstrike: Toward manifesting hidden concurrency typestate bugs. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS XVI, vol. 39, p. 239–250. Association for Computing Machinery, New York, NY, USA (mar 2011), https://doi.org/10.1145/1950365.1950394
32. Gastin, P., Kuske, D.: Uniform satisfiability problem for local temporal logics over Mazurkiewicz traces. Inf. Comput. 208(7), 797–816 (2010)
33. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (1992)
34. Havelund, K., Goldberg, A.: Verify your runs. In: Meyer, B., Woodcock, J. (eds.) Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions. Lecture Notes in Computer Science, vol. 4171, pp. 374–383. Springer (2005)
35. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, Revised Reprint. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2012)
36. 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)
37. Huang, J., Meredith, P.O., Rosu, G.: Maximal sound predictive race detection with control flow abstraction. SIGPLAN Not. 49(6), 337–348 (Jun 2014), https://doi.org/10.1145/2594291.2594315
38. Institute for Software Engineering and Programming Languages: LamaConv - Logics and Automata Converter Library, www.isp.uni-luebeck.de/lamaconv
39. Joshi, P., Sen, K.: Predictive typestate checking of multithreaded java programs. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. p. 288–296. ASE '08, IEEE Computer Society, USA (2008), https://doi.org/10.1109/ASE.2008.39
40. Lamport, L.: A new solution of dijkstra's concurrent programming problem. Commun. ACM 17(8), 453–455 (aug 1974), https://doi.org/10.1145/361082.361093
41. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Commun. ACM 21(7), 558–565 (Jul 1978), https://doi.org/10.1145/359545.359563
42. Lea, D.: A java fork/join framework. In: Proceedings of the ACM 2000 Java Grande Conference, San Francisco, CA, USA, June 3-5, 2000. pp. 36–43 (2000), https://doi.org/10.1145/337449.337465
43. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming 78(5), 293–303 (May 2009)
44. Lodaya, K., Weil, P.: Rationality in algebras with a series operation. Inf. Comput. 171(2), 269–293 (2001)
45. Luo, Q., Rosu, G.: Enforcemop: A runtime property enforcement system for multithreaded programs. In: Proceedings of International Symposium in Software Testing and Analysis (ISSTA'13). pp. 156–166. ACM (July 2013)
46. Manna, Z., Pnueli, A.: A hierarchy of temporal properties (invited paper, 1989). In: Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing. p. 377–410. PODC '90, Association for Computing Machinery, New York, NY, USA (1990), https://doi.org/10.1145/93385.93442

47. 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)
48. Mathur, U., Viswanathan, M.: Atomicity Checking in Linear Time Using Vector Clocks, p. 183–199. Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3373376.3378475
49. 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)
50. Meenakshi, B., Ramanujam, R.: Reasoning about layered message passing systems. Computer Languages, Systems & Structures 30(3-4), 171–206 (2004)
51. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, part I. Theor. Comput. Sci. 13, 85–108 (1981)
52. 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)
53. 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)
54. Rosu, G., Sen, K.: An instrumentation technique for online analysis of multithreaded programs. In: 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings. pp. 268– (2004)
55. Sen, K., Rosu, G., Agha, G.: Runtime safety analysis of multithreaded programs. SIGSOFT Softw. Eng. Notes 28(5), 337–346 (Sep 2003), https://doi.org/10.1145/949952.940116
56. Serbanuta, T., Chen, F., Rosu, G.: Maximal causal models for sequentially consistent systems. In: Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers. pp. 136–150 (2012), https://doi.org/10.1007/978-3-642-35632-2_16
57. Soueidi, C., Falcone, Y.: Artifact Repostiory - Opportunistic Monitoring of Multithreaded Programs (1 2023), https://doi.org/10.6084/m9.figshare.21828570
58. Wang, L., Stoller, S.: Runtime analysis of atomicity for multithreaded programs. IEEE Transactions on Software Engineering 32(2), 93–110 (2006)

Listing 1.1: Readers-writers specification.

```
1  selectors {
2    event AR  on "ReentrantLock.Acquire"  when "%lockname == SharedArr.resourceLock"
3  }
4  types{
5    reader {
6      spawn on "Reader.Run"
7      event read on "SharedArr.read"
8    }
9    writer {
10     spawn on "Writer.Run"
11     event write on "SharedArr.write"
12   }
13 }
14 scope s0 (AR) {
15   property s0r on reader(read)  is "LTL=F(read)"
16   property s0w on writer(write) is "LTL=F(write)"
17
18   atom activereader : count(s0r, T) > 0,
19   atom activewriter : count(s0w, T) > 0,
20   atom onewriter    : count(s0w, T) == 1,
21   atom allreaders   : count(s0r, T) == size(reader)
22
23   check "LTL=G(
24       (activereader XOR activewriter) && (activewriter => onewriter )
25             && (activereader => allreaders)
26     )"
27 }
```

A Expressing Properties Using Existing Tools

We recall from Ex. 6, the properties that we check. Locally we check for an eventual read and write (resp. $\mathbf{F}(\text{read})$ and $\mathbf{F}(\text{write})$), then we count the threads participating, to form the following atomic propositions: activereader, activewriter, allreaders, onewriter. They indicate respectively that: at least one reader, at least one writer, the number of readers that performed a read is equal to the number of reader threads in the program, and exactly one writer performed a write. We recall the scope properties from Ex. 7 they are: mutual exclusion between readers and writers, ensuring that all readers perform a read when at least one does, and mutual exclusion between writers. These properties are all defined on the same scope, the one that alternates between readers and writers (Ex. 3).

Listing 1.1 illustrates the specification for writing local properties and a single scope with the three scope properties. We first define the synchronizing predicates using the *selectors* keyword (Lines 1-3). In this case, we define the predicate AR by looking for an action labeled REENTRANTLOCK.ACQUIRE, and we match its context using the *lockname* key.

After defining the synchronizing predicates, we define the thread types and their associated events (Lines 5-14). For each type we determine the spawn event (Lines 7,11), then all remaining events (Lines 8,12). In this case, we match reading and writing based on the action label. Once thread types are defined, we determine scopes (Line 16-18).

We introduce one scope with the id $\mathrm{s0}$, and specify that its synchronizing predicate is AR (Line 14). Then we determine the local properties, by giving a name for each property (Lines 15-16), determining the thread type and the events it applies on. In this case, each property is expressed using the same string passed to LamaConv [38], which is used to synthesize the monitors. After defining the local property, we define the atomic propositions needed for the scope state (Lines 18-21). For this purpose, we use the helper function count to compute a count of how many monitors (on multiple threads) returned a certain verdict for a local property. Using the function count to summarize the result of local properties, we establish the atomic propositions needed for the scope state. Finally, we introduce the scope property (Lines 23-26), which is generated similarly to a local property but using the atomic propositions for the scope.