Almost Event-Rate Independent Monitoring of Metric Temporal Logic

David Basin[1], Bhargav Nagaraja Bhatt[1], and Dmitriy Traytel[1]

Institute of Information Security, Department of Computer Science, ETH Zürich, Switzerland

Abstract. A monitoring algorithm is trace-length independent if its space consumption does not depend on the number of events processed. The analysis of many monitoring algorithms has aimed at establishing trace-length independence. But a trace-length independent monitor's space consumption can depend on characteristics of the trace other than its size.

We put forward the stronger notion of *event-rate independence*, where the monitor's space usage does not depend on the event rate. This property is critical for monitoring voluminous streams of events arriving at a varying rate. Some previously proposed algorithms for past-only temporal logics satisfy this new property. However, when dealing with future operators, the traditional approach of using a queue to wait for future obligations to be resolved is not event-rate independent. We propose a new algorithm that supports metric past and bounded future operators and is almost event-rate independent, where "almost" denotes a logarithmic dependence on the event rate: the algorithm must store the event rate as a number. We compare our algorithm with traditional ones, providing evidence that almost event-rate independence matters in practice.

1 Introduction

Rules are integral to society. Companies and administrations are highly regulated and subjected to rules, laws, and policies that they must comply to and demonstrate their compliance to. In many domains, the rules are sufficiently precise that automatic monitoring tools can be used to prove compliance or identify violations.

A monitoring tool should solve the standard (*online*) *monitoring problem*: Given a stream of time-stamped data, called events, and a policy formulated in a temporal logic, decide whether the policy is satisfied at every point in the stream [6, 13, 17]. Compared with other verification techniques, the monitoring problem is attractive because it can be solved in a scalable way. Monitoring algorithms usually have a modest time complexity per inspected event. In contrast, keeping the space requirements low for high-velocity event streams is more challenging; this is precisely the problem we tackle here.

Monitoring algorithms have been analyzed in the past with respect to their space requirements. The notion of *trace-length independence* requires a monitor's space complexity to be constant in the overall number of events. In some settings, only algorithms satisfying this property are considered worthy of being called monitors [5]. Trace-length independence aims at distinguishing monitors that can handle huge volumes of data from those that cannot. The classic 3V characterization by volume, velocity, and variety [15], however, tells us that this is only one challenging aspect of big data. Here, we account for another aspect: velocity or event rate.

We propose a new notion, *event-rate independence*, which states that a monitor's space requirement does not depend on the number of events in a fixed time unit. We survey existing monitoring algorithms (Section 2) and identify several for past-only linear temporal logic (ptLTL) [10] and its extension with metric intervals (ptMTL) [19] that have this property. No such monitors exist, however, that support future operators.

We tackle this problem, focusing on *metric temporal logic* (*MTL*) [12] with bounded future operators interpreted over streams of time-stamped events (Section 3). This discrete semantics is based on integer time-stamps, which mirrors the imprecision of physical clocks. A finite number of consecutive events, each defining a *time-point*, might, however, carry the same time-stamp. The event rate is defined as the number of time-points per time-stamp. There are several trace-length independent monitoring algorithms for MTL on streams with a bounded event rate, but none that are event-rate independent or even trace-length independent on streams with an unbounded event rate.

From a traditional standpoint, event-rate independent monitors for MTL seem impossible: future operators require the monitor to wait before it can output a *Boolean verdict* on whether the formula holds. The sheer number of events that the monitor may need to wait for is larger than the event rate. Moreover, it is unclear if one could even achieve a slightly weaker notion, which we call *almost event-rate independence*, where the monitor's space complexity is upper bounded by a logarithm of the event rate (and hence the monitor can store indices or pointers).

As a way out of this dilemma, we propose a monitor that works differently from the traditional ones. Our monitor outputs two kinds of verdicts: standard Boolean verdicts expressing that a formula is true or false at a particular time-point and *equivalence verdicts*. The latter express that the monitor does not know the Boolean verdict at a given time-point, but it knows that the verdict will be equal to another one (presently also not known) at a different time-point. Additionally, our monitor will output verdicts out of order relative to the input stream. Thus, it must indicate in the output to which time-point a verdict belongs. Instead of storing (and outputting) a global time-point reference, we store the time-stamp and the time-point's relative *offset* denoting its position among the time-points labeled with the same time-stamp. We assume that time-stamps can be stored in constant space, which is realistic since 32 bits (as used for Unix time-stamps) will suffice to model seconds for the next twenty years. Storing the offset, however, requires space logarithmic in the event rate.[1] Beyond this, our monitor's space requirement is independent of the event rate.

Although our monitor's output is nonstandard, we are convinced that it is useful. First, the output provides sufficient information to reconstruct all violations. Second, often the monitor's users are only interested in the existence of violations. In this case, they can safely ignore all equivalence verdicts. Third, users are generally interested in the first (earliest) violation. When outputting equivalences, we ensure that the equivalence is output for the later time-points, while the earliest time-point stays in the mon-

[1] One could argue that, if time-stamps model seconds, there is a physical bound on the number of events that fit into this fixed unit of time and the space to store this number can be considered constant. However, we envision applications where time-stamps model days, month, or even years, for which the number of events fitting into one time unit increases dramatically.

itor's memory and is eventually output with a Boolean verdict. Thus, users will always see a truth value at the earliest violating event.

In summary, our work makes the following contributions. We propose the new notion of (almost) event-rate independence, which is crucial for the online monitoring of high-velocity event streams (Section 4). We provide an almost event-rate independent monitoring algorithm for MTL on integer time-stamps with bounded future operators (Section 5). Finally, we report on a prototype implementation of our algorithm (Section 5.4) together with an experimental evaluation (Section 6). Taken together, these contributions lay the foundations for online monitoring that scales both with respect to the volume and the velocity of the event stream.

2 Related Work

There is considerable related work on monitoring. We focus on those algorithms and techniques that are closely related to ours and we touch upon other related works.

Havelund and Roşu [10] propose a simple, yet efficient online monitor for past-time linear temporal logic (ptLTL) using dynamic programming. The satisfaction relation of ptLTL can be recursively defined on a trace by examining the truth-values of subformulas only at the previous time-point. They exploit this insight to develop an algorithm that stores the truth-values of subformulas only at the two latest time-points. The algorithm's space complexity is $\mathcal{O}(n)$, where n is the number of subformulas.

Thati and Roşu [19] extend the results by Havelund and Roşu [10] to provide a trace-length independent, dynamic programming monitoring algorithm for MTL based on derivatives of formulas. Their monitor's space complexity depends only on the size of the formula and the constants occurring in its intervals. Thus their monitor is event-rate independent. However, the algorithm outputs verdicts with respect to a non-standard semantics of MTL, truncated to finite traces. It immediately outputs a verdict at time-points without looking at future events that could possibly alter the verdict. Computing verdicts this way defeats the purpose of (top-level) future operators: An *until* that is not satisfied at the current time-point, but only at the next one, is reported as a violation.

Our algorithm builds on these dynamic programming approaches [10, 19] to handle past-time operators. Our technique for monitoring future formulas under the standard non-truncated semantics of MTL in an event-rate independent manner is new.

Basin et al. [3, 4] introduce techniques to handle MTL and metric first-order temporal logic with bounded future operators, adhering to the standard non-truncated semantics for future formulas. Their monitor uses a queue to postpone evaluation until sufficient time has elapsed to determine the formula's satisfiability at a previous time-point. This requires the algorithm to store in the worst case all time-points during the time-interval it waits. Therefore the monitor's space complexity grows linearly with the event rate, as is confirmed by their empirical evaluation [3, Section 6.3]. Their monitor outputs verdicts in order with respect to time-points, while our algorithm may output verdicts out of order to achieve a better space complexity.

Researchers have developed *trace-length independent* monitoring algorithms for various temporal specification languages. Maler et al. [14] compare the expressive power of timed automata and MTL. They show that past formulas can be converted to

deterministic timed automata (DTA) and there exist future formulas that cannot be represented by a DTA. Ho et al. [11] give a trace-length independent algorithm for MTL in the dense time domain. There exist trace-length independent monitors for timed regular expressions [20], ptLTL extended with counting quantifiers [7], and ptMTL extended with recursive definitions [9]. The underlying logics have different time domains and semantics. We leave the study of event-rate independence in these settings as future work.

3 Metric Temporal Logic

Metric temporal logic (*MTL*) [12] is a logic for specifying qualitative and quantitative temporal properties. We briefly describe the syntax and the point-based semantics of MTL over a discrete time domain. A more in-depth discussion of various flavors of MTL is given elsewhere [4].

Let $\mathbb{I}$ denote the set of non-empty intervals over $\mathbb{N}$. We write an interval in $\mathbb{I}$ as $[a,b]$, where $a \in \mathbb{N}$, $b \in \mathbb{N} \cup \{\infty\}$, $a \leq b$, and $[a,b] = \{x \in \mathbb{N} \mid a \leq x \leq b\}$. For a number $n \in \mathbb{N}$, $I - n$ denotes $\{x - n \mid x \in I\} \cap \mathbb{N}$. For an interval I, let $\mathsf{max}(I)$ denote the largest constant occurring at the endpoints of I, i.e. $\mathsf{max}([a,b]) = b$ if $b \neq \infty$, else a. We write r for the upper bound of the interval, i.e., $\mathsf{r}([a,b]) = b$, which is possibly ∞.

The set of MTL formulas over a set of atomic propositions P is defined inductively:

$$\varphi = p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc_I \varphi \mid \bullet_I \varphi \mid \varphi_1 \, \mathcal{S}_I \, \varphi_2 \mid \varphi_1 \, \mathcal{U}_I \, \varphi_2,$$

where $p \in P$ and $I \in \mathbb{I}$. Along with the standard Boolean operators, MTL includes the temporal operators $\bullet_I$ (*previous*), $\mathcal{S}_I$ (*since*), $\bigcirc_I$ (*next*), and $\mathcal{U}_I$ (*until*), which may be nested freely. We restrict the intervals attached to future operators to be bounded, i.e., we require $\mathsf{r}(I) \neq \infty$, as we want the formulas to be both finitely satisfiable and falsifiable (see [3] for details). We omit the subscript I if $I = [0,\infty)$, and use the usual syntactic sugar for additional Boolean constants and operators $\mathit{true} = p \vee \neg p$, $\mathit{false} = \neg\mathit{true}$, $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$ and future temporal operators *eventually* $\Diamond_I \varphi \equiv \mathit{true} \, \mathcal{U}_I \, \varphi$ and *always* $\Box_I \varphi \equiv \neg\Diamond_I \neg\varphi$ as well as their *past* counterparts *once* $\blacklozenge_I$ and *historically* $\blacksquare_I$.

MTL formulas are interpreted over *streams*, which are infinite sequences of time-stamped events. A time-stamped event is of the form (π_i, τ_i), where $\pi_i \in 2^P$ and $\tau_i \in \mathbb{N}$. Given a stream $\rho = \langle(\pi_0, \tau_0), (\pi_1, \tau_1), (\pi_2, \tau_2), \ldots\rangle$, abbreviated by $\langle(\pi_i, \tau_i)\rangle_{i \in \mathbb{N}}$, we call the τ_i *time-stamps* and their indices i *time-points*. The sequence of time-stamps $\langle\tau_i\rangle_{i \in \mathbb{N}}$ is monotonically increasing, i.e., $\tau_i \leq \tau_{i+1}$ for all $i \geq 0$. Moreover, $\langle\tau_i\rangle_{i \in \mathbb{N}}$ makes progress, i.e., for every $\tau \in \mathbb{N}$, there is some index $i \geq 0$ such that $\tau_i > \tau$. Note that successive time-points can have identical time-stamps; for example, $\langle 5, 5, 5, 7, 8, \ldots\rangle$. Hence, time-stamps may stutter, but only for finitely many time-points. A finite prefix of an event stream is called *trace*.

The semantics of MTL formulas for a given stream $\rho = \langle(\pi_i, \tau_i)\rangle_{i \in \mathbb{N}}$ and a time-point i is defined inductively as follows.

$(\rho, i) \models p$ iff $p \in \pi_i$
$(\rho, i) \models \neg\varphi$ iff $(\rho, i) \not\models \varphi$
$(\rho, i) \models \varphi_1 \vee \varphi_2$ iff $(\rho, i) \models \varphi_1$ or $(\rho, i) \models \varphi_2$
$(\rho, i) \models \bullet_I \varphi$ iff $i > 0$ and $\tau_i - \tau_{i-1} \in I$ and $(\rho, i-1) \models \varphi$
$(\rho, i) \models \bigcirc_I \varphi$ iff $\tau_{i+1} - \tau_i \in I$ and $(\rho, i+1) \models \varphi$

$$(\rho, i) \models \varphi_1 \,\mathcal{S}_I\, \varphi_2 \text{ iff } (\rho, j) \models \varphi_2 \text{ for some } j \le i \text{ with } \tau_i - \tau_j \in I \text{ and } (\rho, k) \models \varphi_1 \text{ for all } j < k \le i$$
$$(\rho, i) \models \varphi_1 \,\mathcal{U}_I\, \varphi_2 \text{ iff } (\rho, j) \models \varphi_2 \text{ for some } j \ge i \text{ with } \tau_j - \tau_i \in I \text{ and } (\rho, k) \models \varphi_1 \text{ for all } i \le k < j$$

When the stream ρ is clear from the context, we also simply write $i \models \varphi$.

From the semantics of MTL, it is easy to derive an equivalent recursive definition for the *until* and *since* operators for a fixed stream ρ:

$$i \models \varphi_1 \,\mathcal{S}_I\, \varphi_2 \text{ iff } 0 \in I \text{ and } i \models \varphi_2, \text{ or } i > 0, \tau_i - \tau_{i-1} \le \mathsf{r}(I), i \models \varphi_1, \text{ and } i-1 \models \varphi_1 \,\mathcal{S}_{I-(\tau_i-\tau_{i-1})}\, \varphi_2$$
$$i \models \varphi_1 \,\mathcal{U}_I\, \varphi_2 \text{ iff } 0 \in I \text{ and } i \models \varphi_2, \text{ or } \tau_{i+1} - \tau_i \le \mathsf{r}(I), i \models \varphi_1, \text{ and } i+1 \models \varphi_1 \,\mathcal{U}_{I-(\tau_{i+1}-\tau_i)}\, \varphi_2$$

Note that the formula being "evaluated" on the right-hand side of these recursive equations has the same structure as the initial formula, except that the interval has been shifted by the difference between the current and the previous (or the next) time-stamps. Our algorithm, described in Section 5, uses these recursive equations to update the monitor's state by simultaneously monitoring the formulas arising from all possible interval shifts. We call such formulas *interval-skewed subformulas*. For an MTL formula φ, let $\mathsf{SF}(\varphi)$ denote the set of its subformulas defined in the usual manner. Note that $\varphi \in \mathsf{SF}(\varphi)$. The set of interval-skewed subformulas of φ is defined as

$$\mathsf{ISF}(\varphi) = \mathsf{SF}(\varphi) \cup \{\varphi_1 \,\mathcal{S}_{I-n}\, \varphi_2 \mid \varphi_1 \,\mathcal{S}_I\, \varphi_2 \in \mathsf{SF}(\varphi) \text{ and } n \in [1, \mathsf{max}(I)]\} \cup \{\varphi_1 \,\mathcal{U}_{I-n}\, \varphi_2 \mid \varphi_1 \,\mathcal{U}_I\, \varphi_2 \in \mathsf{SF}(\varphi) \text{ and } n \in [1, \mathsf{max}(I)]\}.$$

Clearly, the size of $\mathsf{ISF}(\varphi)$ is bounded by $\mathcal{O}(|\mathsf{SF}(\varphi)| \times c)$, where c is the largest integer constant occurring in the intervals of φ. We define a well-order $<$ over $\mathsf{ISF}(\varphi)$ that respects the following conditions:

- if φ_1 is a subformula of φ_2 and $\varphi_1 \neq \varphi_2$, then $\varphi_1 < \varphi_2$
- if $\varphi_1 = \alpha \,\mathcal{S}_I\, \beta$ and $\varphi_2 = \alpha S_{I'} \beta$ and $I' = I - n$ for some $n > 0$, then $\varphi_1 < \varphi_2$.

We use this to order the elements of $\mathsf{ISF}(\varphi)$ into an array in Section 5.

We also define the *future reach* (FR) of an MTL formula following Ho et al. [11], which we subsequently use to analyze the complexity of our proposed algorithm.

$$\mathsf{FR}(p) = 0 \qquad \mathsf{FR}(\neg\varphi) = \mathsf{FR}(\varphi) \qquad \mathsf{FR}(\varphi_1 \vee \varphi_2) = \mathsf{max}(\mathsf{FR}(\varphi_1), \mathsf{FR}(\varphi_2))$$
$$\mathsf{FR}(\bullet_I \varphi) = \mathsf{FR}(\varphi) - \mathsf{inf}(I) \qquad \mathsf{FR}(\bigcirc_I \varphi) = \mathsf{sup}(I) + \mathsf{FR}(\varphi)$$
$$\mathsf{FR}(\varphi_1 \,\mathcal{S}_I\, \varphi_2) = \mathsf{maximum}(\mathsf{FR}(\varphi_1), \mathsf{FR}(\varphi_2) - \mathsf{inf}(I))$$
$$\mathsf{FR}(\varphi_1 \,\mathcal{U}_I\, \varphi_2) = \mathsf{sup}(I) + \mathsf{maximum}(\mathsf{FR}(\varphi_1), \mathsf{FR}(\varphi_2))$$

Here maximum denotes the maximum of two integers and sup and inf denote the *supremum* and *infimum* of sets of integers, respectively. For a bounded future MTL formula φ, we have $\mathsf{FR}(\varphi) \neq \infty$. Intuitively, events that have a time-stamp larger than $\tau_i + \mathsf{FR}(\varphi)$ are irrelevant for determining φ's validity at a time-point i with time-stamp τ_i.

Example 1. Consider the formula $\varphi = a \,\mathcal{U}_{[0,1]}\, b$ and the event stream $\rho = \langle(\{a\}, 1), (\{a\}, 2), (\{a\}, 2), (\{b\}, 3), (\{a, b\}, 4), \ldots\rangle$. In Figure 1, $\top$ and $\bot$ denote the satisfaction and violation of φ. Note that the verdict $\bot$ at time-point 0 is determined only after the event $(\{b\}, 3)$ has arrived. This observation would also apply, even if the event $(\{a\}, 2)$ was replicated arbitrarily often in the stream.

i (time-point)	0	1	2	3	4	...
π_i (events)	$\{a\}$	$\{a\}$	$\{a\}$	$\{b\}$	$\{a,b\}$	...
τ_i (time-stamps)	1	2	2	3	4	...
$i \models a\,\mathcal{U}_{[0,1]}\,b$	$\bot$	$\top$	$\top$	$\top$	$\top$	...

Fig. 1. Evaluation of $a\,\mathcal{U}_{[0,1]}\,b$ on an example stream

4 Almost Event-Rate Independence

The space complexity of monitoring algorithms has been previously analyzed with respect to two parameters: *formula size* and *trace length*. In most scenarios, the formula is much smaller than the trace and does not change during monitoring. Hence, an algorithm with a space complexity exponential in the formula size is usually tolerable, but a space complexity linear in the trace length is problematic since this corresponds to storing the entire trace. Recently, researchers have studied *trace-length independence* [5]. A monitor is trace-length independent if its efficiency does not decline as the number of events increases. In the setting of MTL, we call a monitoring algorithm $\mathcal{M}$ *trace-length independent on the stream* ρ if the space required by $\mathcal{M}$ to output the verdict at time-point i when monitoring ρ is independent of i. This property is critical for determining whether a monitor scales to large quantities of data. However, it does not yield insights into the monitor's performance regarding other aspects of the stream such as its velocity.

We propose the notion of event-rate independence, which not only guarantees the monitor's memory efficiency with respect to the number of events, but also with respect to the rate at which the events arrive. A varying event rate is a realistic concern in many practically relevant monitoring scenarios. For example, if the unit of time-stamps is on the order of days, there may be millions of time-points with the same time-stamp in a stream. An event-rate dependent algorithm may work well on days with a few thousand events, but fall short of memory when the number of events rises significantly. (Such a situation could be an indicator that something interesting happened, which in turn makes the monitor's output particularly valuable on that day.)

We first formally define a stream's *event rate*.

Definition 1. The *event rate* er of a stream $\rho = \langle(\pi_i, \tau_i)\rangle_{i\in\mathbb{N}}$ at time-stamp τ is defined as the number of time-points whose time-stamps are equal to τ, i.e., $\mathsf{er}_\rho(\tau) = |\{i \mid \tau_i = \tau\}|$.

An online monitoring algorithm $\mathcal{M}$ for MTL is *event-rate independent on the stream* ρ if for all time-points i the monitor $\mathcal{M}$'s space complexity to compute the verdict at i is constant with respect to $\mathsf{er}_\rho(\tau_j)$ for all $j \leq i$, i.e., the event rates in ρ at all time-stamps up to and including the current one. Ultimately, we are interested in monitors that are event-rate independent on all streams ρ. For example, the dynamic programming algorithms [10, 19] are event-rate independent on all streams ρ for past-only MTL.

The trace length up to time-point i is greater than the sum of the event rates $\mathsf{er}_\rho(\tau)$ for $\tau < \tau_i$ for all streams ρ. Hence, we obtain the following lemma by contraposition.

Lemma 1. *Fix a stream ρ. Let $\mathcal{M}$ be a monitoring algorithm for MTL. If $\mathcal{M}$ is event-rate independent on ρ, then $\mathcal{M}$ is trace-length independent on ρ.*

In general, event-rate independence is not strictly stronger than trace-length independence. To see this, consider the following stream where the event rate itself depends on

the trace length: $\rho = \langle(\pi_0, 0), (\pi_1, 1), (\pi_1, 1), (\pi_2, 2), (\pi_2, 2), (\pi_2, 2), (\pi_2, 2), \ldots\rangle$, where (π_τ, τ) is repeated 2^τ times. Any event-rate dependent monitor for ρ is also trace-length dependent, since the event rate is roughly half of the trace length at each time-point.

In contrast to the above example, streams arising in practice have a bound on the event rate. For such an (*event-rate*) *bounded stream* ρ we have $\forall i.\ \mathsf{er}_\rho(\tau_i) < b_\rho$ for some arbitrary but fixed b_ρ. In fact, the related *bounded variability* assumption [8, 11, 14] is deemed necessary for trace-length independence. The consideration of the event rate clarifies the need for this assumption: On bounded streams ρ, event-rate independence is strictly stronger than trace-length independence. For example, monitors using a waiting queue for future operators [3] are trace-length independent on ρ, but not event-rate independent on ρ. On unbounded streams, i.e., streams that are not event-rate bounded, the two notions coincide. This is in line with the fact that there are trace-length independent monitors for MTL (with future operators) on bounded streams [3, 11], but none on unbounded streams.

Event-rate independence and trace-length independence for unbounded streams are indeed impossible if we adhere to the mode of operation of existing MTL monitors. Existing monitors output verdicts *monotonically*, i.e., for time-points i and j, if $i < j$ then the verdict at i is output before the verdict at j. Monotonicity makes any monitor handling future operators linearly event-rate dependent (and hence trace-length dependent for unbounded streams), as it must wait for and therefore store information associated to more than $\mathsf{er}_\rho(\tau)$-many events (for some τ) before being able to output a verdict. So event-rate independence seems to be too strong a condition for traditional monitors.

To overcome this problem, our monitor outputs verdicts differently. In addition to the standard Boolean verdicts $\top$ and $\bot$, it outputs *equivalence verdicts* $j \equiv i$ (with $i < j$) if it is certain that the verdict at time-point j will be equivalent to the verdict at a previous time-point i, even if the exact truth value is presently unknown at both points. This makes verdict outputs *non-monotonic with respect to time-points*, but it is still possible to ensure *monotonicity with respect to time-stamps* for time-stamps that are far enough apart. More precisely, a monitor that is monotonic with respect to time-stamps outputs the verdict at i before the verdict at j when monitoring φ, if $\tau_j - \tau_i > \mathsf{FR}(\varphi)$.

To output equivalence verdicts, the algorithm must refer to time-points. This requires non-constant space, e.g., logarithmic space for natural numbers. Time-points increase with the trace length, leading to a logarithmic dependence on the trace length. An alternative way to refer to time-points is to use time-stamps together with an offset pointing into a block of consecutive time-points labeled with the same time-stamp. (The size of such a block is bounded by the event rate.) The space requirement of an algorithm outputting such verdicts is therefore not event-rate independent. However, it is logarithmic in the event rate. These observations suggest the slightly weaker notion of almost event-rate independence, which is defined identically to event-rate independence except that the space complexity is upper bounded by a logarithm of the event rate.

Definition 2. An online monitoring algorithm $\mathcal{M}$ for MTL is *almost event-rate independent* if for all time-points i and streams ρ the space complexity of $\mathcal{M}$ for outputting the verdict at i is $\mathcal{O}(\log(\max_{j \le i} \mathsf{er}_\rho(\tau_j)))$.

Our proposed monitor is almost event-rate independent. Moreover, it is the first almost trace-length independent monitor on unbounded streams.

5 Monitoring Algorithm

We describe the high-level design of our monitoring algorithm for MTL informally. Then we give a formal description using functional programming notation, prove its correctness and almost event-rate independence, and discuss implementation details.

5.1 Informal Account

The idea of outputting equivalence verdicts draws inspiration from a natural way to approach simultaneous suffix matching with automata. To decide which suffixes of a word are matched by an automaton, a naive approach is to start running the automaton at each position in the word. For a word of length n this requires storing n copies of the automaton. A more space-efficient approach is to store a single copy, and use markers (one marker for each position in the word) that are moved between states upon transitions. If n is larger than the number of states, then at some point two markers will necessarily mark the same state. At this point, it suffices to output their equivalence and track only one of them, since they would travel through the automaton together. Our algorithm follows a similar approach; however, we avoid explicitly constructing automata from formulas.

Our algorithm builds on Havelund and Roşu's dynamic programming algorithm for past-time LTL [10], where the monitor's state consists of an array of Boolean verdicts for all subformulas of the monitored formula at a given time-point. The array is dynamically updated when consuming the next event based on the recursive definition of satisfiability for LTL. To support intervals, we use the idea by Thati and Roşu [19] to store an array of verdicts for all interval-skewed subformulas instead of plain subformulas as in Havelund and Roşu. This accounts for possible interval changes when moving between different time-stamps according to the recursive definition of satisfiability for past-time MTL. This step crucially relies on the time-stamps being integer-valued, as otherwise the number of skewed subformulas would be infinite.

The problem with future operators is that they require us to wait until we are able to output a verdict. At first, we sidestep almost event-rate independence and formulate a dynamic programming algorithm that treats past operators as Havelund and Roşu's algorithm [10] but also supports future operators. The recursive equation for *until* reduces the satisfaction of a formula $\varphi_1 \,\mathcal{U}_I\, \varphi_2$ at the current time-point to a Boolean combination of the satisfaction of φ_1 and φ_2 at the current time-point and the satisfaction of $\varphi_1 \,\mathcal{U}_{I-n}\, \varphi_2$ (for some n) at the next time-point. While we can immediately resolve the dependencies on the current time-point, those on the next time-point force us to wait. This also means that we cannot store the verdict in an array (because we do not know it yet), but instead we will store the dependency in the form of pointers to some entries in the next array to be filled. In general, our dynamically updated array (of length $|\mathsf{ISF}(\varphi)|$), indexed by interval-skewed subformulas, will contain Boolean expressions instead of Booleans, in which the variables denote the dependencies on those next entries.

Additionally, we may only output verdicts when the Boolean expressions are resolved to a Boolean verdict. This will happen eventually, since in our setting time progresses and future intervals are bounded. But until this happens, the yet-to-be-output

Boolean expressions must be stored, which affects the algorithm's space consumption. In the worst case, the monitor would store as many expressions as there are time-points in any interval of timespan d, where d is the future reach of the monitored formula.

Finally, to obtain almost event-rate independence, we refine our monitor's output by allowing it to output equivalence verdicts between different time-points. As soon as the monitor sees two semantically equivalent Boolean expressions, it may output such verdicts and discard one of the two expressions. Since there are only $\mathcal{O}(2^{2^{|\mathsf{ISF}(\varphi)|}})$ semantically different Boolean expressions in $\mathcal{O}(|\mathsf{ISF}(\varphi)|)$ variables (corresponding to the verdicts for interval-skewed subformulas at the next time-point), the space required to store them depends only on the monitored formula φ. However, for the equivalence verdicts to be understandable to users, the equivalences must refer to different time-points via indices. Storing those indices requires logarithmic space in the event rate. Hence, the overall algorithm is almost event-rate independent.

5.2 The Algorithm

We now give a more formal description of our algorithm. For the presentation, we use a functional programming-style pseudo code, with pattern matching, that resembles Standard ML. Type constructors, such as _ *list* or _ *array* for functional lists and arrays (lists of fixed length with constant time element access), are written postfix, with the exception of the product type $\times$ and the function space $\to$, which are written infix. We write $\mathbb{N}$ for the type of natural numbers and $\frownie$ for the type of time-stamps (although, in our case, these are again just natural numbers). Lists are either empty $[]$ or constructed by prepending an element to a list $x :: xs$. List concatenation is written infix as $+\!\!+$. Anonymous functions are introduced using λ-abstractions.

Our monitor for a fixed formula Φ operates on an input stream of time-stamped events I and writes verdicts to an output stream O. Additionally, it starts in some initial state init of type σ and can perform state transitions $\mathsf{step} : \sigma \to \sigma$. The state consists of three parts: a list of time-stamped Boolean expressions for which the verdict depends on future events, a current time-stamp, and an array of Boolean expressions for all interval-skewed subformulas at the current time-point (similarly to the state of Havelund and Roşu's algorithm). Expressions for small subformulas are stored at low indices in this array, while the monitored formula Φ has index $|\mathsf{ISF}(\Phi)| - 1$. In other words, if we think of the array as being indexed by subformulas, then the array's indices are ordered by the well-order $<$. We formalize the state using a record type:

$$\text{record}\ \sigma = \{\mathsf{hist} : (\frownie \times \mathbb{N} \times \mathit{bexp})\ \mathit{list}, \mathsf{now} : \frownie \times \mathbb{N}, \mathsf{arr} : \frownie \to \mathit{bexp}\ \mathit{array}\}.$$

Two points are worth noting here. First, in addition to the time-stamp for each time-point, we store an *offset* of type $\mathbb{N}$, which stores the position of the time-point within a block of time-points with the same time-stamp. Using the time-stamp and the offset, each time-point can be uniquely identified. Second, the array in arr has a dependency on a future time-stamp because the recursive definition of satisfaction for *until* depends the time-stamp difference between the next and the current time-point. As a result, our monitor will output a verdict for a time-point only after having seen the time-stamp of the next time-point. We will revisit and rectify this limitation in Section 5.4.

Overloading notation, (*Boolean*) *expressions* can be defined inductively as follows:

```
init = {hist = [], now = (−1, 0), arr = λ_. ⊥^n}
step {hist = h, now = (τ, i), arr = fa} =
  let (π, τ') ⇐ I
      a = fa τ'
      h' = fold (update a) (rev h) []
      j = if τ = τ' then i + 1 else 0
  in  {hist = add (τ, i, a[Φ]) h', now = (τ', j),
       arr = progress a τ π τ'}

update a (τ, i, b) h =
  let c = subst (λx. a[x]) b
  in  if c = ⊤ ∨ c = ⊥
      then
        let if τ ≥ 0 then (τ, i, c) ⇒ O
        in  h
      else
        add (τ, i, c) h
```

Fig. 2. The transition system of the monitor: init and step

$$bexp = \bot \mid \top \mid bexp \wedge bexp \mid bexp \vee bexp \mid \neg bexp \mid \mathsf{var}\ \mathbb{N}.$$

Here, a variable should be thought of as a pointer into the arr array of the yet-to-be-computed next state, i.e., a natural number less than n, where n is the number of interval-skewed subformulas of Φ. To lighten the notation, we implicitly convert interval-skewed subformulas of Φ to natural numbers between 0 and $n-1$, and vice versa. For example, we write var φ (or $a[\varphi]$) to denote a variable pointing to the array entry corresponding to the formula φ (or the array entry itself). We assume that all expressions of type *bexp* are normalized using Boolean simplifications, e.g., $\bot \wedge x$ is rewritten to $\bot$. Thus, each expression is either a Boolean $\bot$ or $\top$ or does not contain $\bot$ or $\top$ as a subexpression. Furthermore, we will use the function subst : $(\mathbb{N} \to bexp) \to bexp \to bexp$ to replace variables with expressions according to the given function argument as well as a decision procedure $\equiv : bexp \to bexp \to \{\bot, \top\}$ for the semantic equivalence of Boolean expressions. We omit the definitions of those two functions.

The monitor's initial state init and its transition function step are shown in Figure 2. The function step formalizes the transition from the current time-point to the next one. First, it retrieves the new event π and its time-stamp τ' from the input stream I (which we write as $(\pi, \tau') \Leftarrow I$). Using τ', the next step evaluates the future-dependent array fa to obtain an array of Boolean expressions a. Note that the expressions in a refer to the array of the next state, while all expressions in the history h refer to the current state, namely to a itself. To overcome this mismatch, the monitor iterates over the history using the standard fold combinator on lists and updates each of the Boolean expressions to refer to the next state using subst in the function update. This update may convert some of the expressions into Boolean verdicts, which are immediately output (written $\ldots \Rightarrow O$) and removed from the history. Next, the monitor computes the new offset j depending on whether the time-stamp has increased. Finally, the last entry of the array a is added to the history (or output in case it is a Boolean verdict) using the function add and the new future-dependent array is produced by (a partial application of) the progress function and stored in the state. We describe these two core functions next.

We consider three different implementations of the add function:

$$\mathsf{add}\ (x\ \mathsf{as}\ (\_, \_, c))\ xs = \text{if } c = \bot \vee c = \top \text{ then } (\text{let } x \Rightarrow O \text{ in } xs) \text{ else } \begin{cases} x :: xs & \text{NAIVE} \\ \mathsf{go}\ \bot\ []\ x\ xs & \text{GLOBAL} \\ \mathsf{go}\ \top\ []\ x\ xs & \text{LOCAL} \end{cases}$$

```
progress a τ π τ′ τ″ =
  let b = ⊥^n
    for x = 0, …, n − 1
      b[x] = case x of
        | p       ⇒ p ∈ π
        | ¬φ      ⇒ ¬ b[φ]
        | φ ∨ ψ   ⇒ b[φ] ∨ b[ψ]
        | ●_I φ   ⇒ if τ′ − τ ∈ I then subst (λx. b[x]) a[φ] else ⊥
        | ○_I φ   ⇒ if τ″ − τ′ ∈ I then var φ else ⊥
        | φ S_I ψ ⇒ (if 0 ∈ I then b[ψ] else ⊥) ∨
                    (if τ′ − τ ≤ r(I) then b[φ] ∧ subst (λx. b[x]) a[φ S_{I−(τ′−τ)} ψ] else ⊥)
        | φ U_I ψ ⇒ (if 0 ∈ I then b[ψ] else ⊥) ∨
                    (if τ″ − τ′ ≤ r(I) then b[φ] ∧ var (φ S_{I−(τ″−τ′)} ψ) else ⊥)
  in b

go loc done x [] = x :: rev done
go loc done (τ, i, b) ((τ′, j, c) :: todo) =
  if loc ∧ τ ≠ τ′ then (τ, i, b) :: rev done ++ (τ′, j, c) :: todo
  else if c ≡ d then
    let (τ, i) ≡ (τ′, j) ⇒ O in rev done ++ (τ′, j, c) :: todo
  else go loc ((τ′, j, c) :: done) (τ, i, b) todo
```

Fig. 3. Recursive formula progression and insertion modulo semantic expression equivalence

The NAIVE version simply prepends the element to the history (which is kept in reversed order with respect to the input stream). This version is not almost event-rate independent. The GLOBAL version adds the new expression only if there is no semantically equivalent expression in the history. The LOCAL version adds the new expression only if there is no semantically equivalent expression labeled with the same time-point. Whenever an expression is *not* added to the history, an equivalence verdict is output. Both versions, LOCAL and GLOBAL, are implemented using the auxiliary function go shown in Figure 3 and give rise to almost event-rate independent algorithms.

The last missing piece is the update of the arr entry of the monitor's state. The function progress shown in Figure 3 performs this update. It has access to the previous time-stamp τ, the current time-stamp τ', the next time-stamp τ'', the current event π, and the previous array of Boolean expressions a. Given these inputs, it fills the next array b starting from the smallest subformulas and progressing up to the formula Φ itself. Each array entry is filled following the recursive definition of satisfaction of the topmost operator of the formula it corresponds to. Moreover, whenever the previous array a is accessed for past operators, the retrieved expression's dependencies are updated using subst as before. In contrast, for future dependencies, the var constructor of expressions is used.

Example 1. (continued) Figure 4 shows the internal states of the GLOBAL version of our algorithm when monitoring the formula $a\,\mathcal{U}_{[0,1]}\,b$ on the stream $\rho = \langle(\{a\},1),(\{a\},2),(\{a\},2),(\{b\},3),(\{a,b\},4),\ldots\rangle$. The first two rows show the incoming events and their time-stamps, the third the within-time-stamp offset, and the fourth the current history. The next four rows are dedicated to the Boolean expressions stored for each interval-skewed subformula. The last row displays the monitor's verdicts. At each time-point, the monitor's state consists (roughly) of one column from this table. Since it is hard to display the function *fa*, we show instead the result of applying *fa* to the time-stamp of the next state. This causes a delay of one time-point between the values in the arrays and the history updates and verdict outputs.

π		$\{a\}$	$\{a\}$	$\{a\}$	$\{b\}$	$\{a,b\}$
τ	$-$	1	2	2	3	4
i	0	0	0	1	0	0
h	[]	[]	$[(1, 0, \mathsf{var}\,\varphi_0)]$	$[(2, 0, \mathsf{var}\,\varphi_1), (1, 0, \mathsf{var}\,\varphi_0)]$	$[(2, 0, \mathsf{var}\,\varphi_0)]$	[]
	$fa\,1$	$fa\,2$	$fa\,2$	$fa\,3$	$fa\,4$	$\cdots$
a	$\bot$	$\top$	$\top$	$\top$	$\bot$	$\cdots$
b	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\cdots$
$\varphi_0 = a\,\mathcal{U}_{[0,0]}\,b$	$\bot$	$\bot$	$\mathsf{var}\,\varphi_0$	$\bot$	$\top$	$\cdots$
$\varphi_1 = a\,\mathcal{U}_{[0,1]}\,b$	$\bot$	$\mathsf{var}\,\varphi_0$	$\mathsf{var}\,\varphi_1$	$\mathsf{var}\,\varphi_0$	$\top$	$\cdots$
verdicts					$(1,0) = \bot$ $(2,1) = (2,0)$	$(2,0) = \top$ $(3,0) = \top$

Fig. 4. An execution of the monitoring algorithm on $a\,\mathcal{U}_{[0,1]}\,b$

5.3 Correctness and Complexity Analysis

In this subsection, we fix a formula Φ and a stream ρ. To prove the soundness and completeness of our monitor and to establish its space complexity bounds, we formulate an invariant $\mathcal{I}$ that holds after processing the first event and all subsequent states.

$$
\begin{aligned}
&\mathcal{I}\,\{\mathsf{hist} = h, \mathsf{now} = (\tau, i), \mathsf{arr} = fa)\} = \\
&\quad (\mathcal{I}1) \quad (\forall(\tau', j, b) \in h.\ \tau' @\, j \models \Phi \leftrightarrow \tau @ i \models_{bexp} b) \\
&\wedge\,(\mathcal{I}2) \quad (\forall \varphi \in \mathsf{ISF}(\Phi).\ \tau @ i \models \varphi \leftrightarrow \tau @ i + 1 \models_{bexp} fa\ (\tau_{\tau @ i+1})[\varphi]) \\
&\wedge\,(\mathcal{I}3) \quad (\forall \varphi \in \mathsf{ISF}(\Phi).\ \mathsf{vars}\ (fa\ (\tau_{\tau @ i+1})[\varphi]) \subseteq \mathsf{ISF}(\varphi)) \\
&\wedge\,(\mathcal{I}4) \quad (\forall(\tau', j, b) \in h.\ b \neq \top \wedge b \neq \bot) \\
&\wedge\,(\mathcal{I}5) \quad h \text{ is sorted in strictly descending order by time-point} \\
&\wedge\,(\mathcal{I}6) \quad (\forall(\tau', j, b) \in h.\ \forall(\tau'', k, c) \in h.\ \tau' @\, j \neq \tau'' @ k \rightarrow \mathsf{compact}\ \tau'\ \tau''\ b\ c)
\end{aligned}
$$

We write $\tau @ i$ to denote the time-point uniquely identified by the time-stamp τ and the within-time-stamp offset i. Moreover, vars is the set of vars in a Boolean expression, τ_k is the time-stamp from ρ at time-point k, and $\models_{bexp}$ is the lifting of MTL satisfaction to expressions. For the base case of this lifting, we have $k \models_{bexp} \mathsf{var}\,\varphi \leftrightarrow k \models \varphi$.

The invariant consists of six predicates. $(\mathcal{I}1)$ and $(\mathcal{I}2)$ capture the semantics of the entries in the history and the expression array. $(\mathcal{I}3)$ expresses that future dependencies in any expression indexed by a subformula φ may only refer to φ's interval-skewed subformulas. $(\mathcal{I}4)$ and $(\mathcal{I}5)$ are important structural properties of the history. $(\mathcal{I}6)$ is crucial for our complexity analysis. It uses an auxiliary predicate compact, defined differently for each of the three versions of the monitoring algorithm we consider.

$$
\mathsf{compact}\ \tau'\ \tau''\ b\ c = \begin{cases} \top & \text{NAIVE} \\ b \not\equiv c & \text{GLOBAL} \\ \tau' = \tau'' \rightarrow b \not\equiv c & \text{LOCAL} \end{cases}
$$

We prove that $\mathcal{I}$ holds for every reachable state except the initial state itself. In the initial state $(\mathcal{I}2)$ is violated. The fa array of the initial state is accessed only for past-time operators at the first event. In this case, the stored values $\bot$ for all subformulas have exactly the right semantics: essentially they affirm that there is no previous time-point.

Lemma 2. *$\mathcal{I}$ (step init) and for any state s if $\mathcal{I}(s)$ then $\mathcal{I}$ (step s)*

Proof (core idea). The core of the proof is the preservation of $(\mathcal{I}2)$ by the progress function. We prove the following auxiliary lemma: *Fix a stream $\rho = \langle(\pi_i, \tau_i)\rangle_{i\in\mathbb{N}}$ and a time-point k. Assume* progress $a\ \tau_k\ \pi_{k+1}\ \tau_{k+1}\ \tau_{k+1} = b$ *and for all $\varphi \in \mathsf{ISF}(\Phi)$ we have $k \models \varphi \leftrightarrow k+1 \models_{bexp} a[\varphi]$. Then $k+1 \models \varphi \leftrightarrow k+2 \models_{bexp} b[\varphi]$ holds for all $\varphi \in \mathsf{ISF}(\Phi)$.*

The lemma follows by well-founded induction on the lexicographic product of the natural number order on time-points and the order $<$ on formulas: Fix $\varphi \in \mathsf{ISF}(\Phi)$. The induction hypothesis allows us to assume $k+1 \models \psi \leftrightarrow k+2 \models_{bexp} b[\psi]$ for any $\psi < \varphi$. We continue by a case distinction on φ and present here only the case where $\varphi = \varphi_1\, \mathcal{U}_I\, \varphi_2$. Let $\Delta = \tau'' - \tau'$ and $I' = I - \Delta$. We calculate

$$
\begin{aligned}
k+1 \models \varphi_1\, \mathcal{U}_I\, \varphi_2 \quad &\overset{\text{recursive def. of } \models}{\longleftrightarrow} \quad (0 \in I \wedge k+1 \models \varphi_2) \vee \\
&\qquad (\Delta \leq \mathsf{r}(I) \wedge k+1 \models \varphi_1 \wedge k+2 \models \varphi_1\, \mathcal{U}_{I'}\, \varphi_2) \\
&\overset{\text{twice IH + def. } \models_{bexp}}{\longleftrightarrow} (0 \in I \wedge k+2 \models_{bexp} b[\varphi_2]) \vee \\
&(\Delta \leq \mathsf{r}(I) \wedge k+2 \models_{bexp} b[\varphi_1] \wedge k+2 \models_{bexp} \mathsf{var}\ (\varphi_1\, \mathcal{U}_{I'}\, \varphi_2)) \\
&\overset{\text{def. of progress}}{\longleftrightarrow} \quad k+2 \models_{bexp} b[\varphi_1\, \mathcal{U}_I\, \varphi_2]
\end{aligned}
$$

Other cases follow similarly. Past operators additionally use the assumption on a. □

The step from the invariant to a correctness theorem is easy. For soundness, we calculate the expected semantic properties for verdicts output in a step taking $(\mathcal{I}1)$ and $(\mathcal{I}2)$ of the invariant into account. Completeness also holds: for each time-point either a verdict is output or an expression is inserted into the history. Each expression from the history is eventually output as time progresses and all future intervals are bounded.

Theorem 1 (Correctness). *The monitor for a formula Φ is sound: whenever it outputs the Boolean verdict (τ, i, b) we have $\tau@i \models \Phi \leftrightarrow b$ and whenever it outputs the equivalence verdict $(\tau, i) \equiv (\tau', j)$ we have $\tau@i > \tau'@j$ and $\tau@i \models \Phi \leftrightarrow \tau'@j \models \Phi$. For the* LOCAL *mode, we additionally have $\tau = \tau'$. Moreover, the monitor is complete.*

Finally, we establish complexity bounds. Let $n = |\mathsf{ISF}(\Phi)|$ and $d = \mathsf{FR}(\varphi)$. Note that $d \leq n$. The size of a Boolean expression in n variables can be bounded by 2^n assuming a normal form for expressions such as CNF. Then the size of the future-dependent array arr is $n \cdot 2^n$. The length of the history depends on the version of the algorithm used and (except for the NAIVE algorithm) dominates the size of arr.

Theorem 2 (Space Complexity). *The space complexity for storing all Boolean expressions used by the three versions of the algorithm at the time-stamp τ is*

NAIVE: $\mathcal{O}(2^n \cdot (n + \sum_{\tau'=\tau-d}^{\tau} \mathsf{er}(\tau')))$, GLOBAL: $\mathcal{O}(2^{2^n+n})$, *and* LOCAL: $\mathcal{O}(d \cdot 2^{2^n+n})$.

Time-stamps additionally require a constant and the offsets a logarithmic amount of space in the event rate. Hence, GLOBAL *and* LOCAL *are almost event-rate independent.*

Proof. Each stored Boolean expression requires $\mathcal{O}(2^n)$ space. The bound for NAIVE follows since, at time-stamp τ, we can output Boolean verdicts for all time-stamps that are at most $\tau - d$. Hence, the history needs to store only those expressions that fit into the interval $(\tau - d, \tau]$. For GLOBAL (or LOCAL) there are at most 2^{2^n} (or $d \cdot 2^{2^n}$) semantically different Boolean expressions that must be stored in the history. □

5.4 Implementation

We have implemented the presented algorithm using Standard ML. The implementation comprises just roughly 600 lines of code. It is available online [1].

Our implementation follows the pseudo-code in Section 5.2. In one aspect, it takes a more refined approach. The monitor's users would like violations to be reported as early as possible. The presented monitor does not do this as it delays the output of verdicts for one time-point, even if no future operators are involved. Our implementation improves this by refining the type of arr in the monitor's state from $\clock \times \mathbb{N} \times (\clock \to \mathit{bexp}\ \mathit{array})$ to the more precise $\clock \times \mathbb{N} \times \mathit{bexp}_\mathsf{f}\ \mathit{array}$, where the type of *potentially future expressions* bexp_f is either an immediate Boolean expression or a future-dependent expression as before. Formally $\mathit{bexp}_\mathsf{f} = \mathsf{Now}\ \mathit{bexp} \mid \mathsf{Later}\ (\clock \to \mathit{bexp})$.

This refined type makes it possible to output verdicts at the current time-point instead of the following one, provided that the computation of progress resulted in a Now constructor for the monitored formula Φ. Accordingly, the function progress must be refined to carefully assemble possibly future expressions to maximize the number of Now constructors in the array. To achieve this, all constructors (e.g., $\wedge$) of *bexp* are lifted to functions (e.g., $\wedge_\mathsf{f}$) on bexp_f that try to produce as many Nows as possible by applying simplification rules such as $\mathsf{Now}\ \bot \wedge_\mathsf{f} \mathsf{Later}\ f = \mathsf{Now}\ \bot$.

To implement the expression equivalence check, we use a simple BDD based algorithm that has been formally verified in the Isabelle proof assistant by Nipkow [16]. It would be interesting to explore working with BDDs instead of Boolean expressions all the time (and not only in the equivalence check) to potentially improve time complexity.

6 Evaluation

We compare the three versions of our tool with MONPOLY [2, 3], a state-of-the art monitor for *metric first-order temporal logic*. The experiments were run on a 3.1 GHz dual-core Intel Core-i7 processor and 16 GB RAM. We evaluate the memory consumption of all tools while monitoring four MTL formulas on pseudo-randomly generated event logs with varying average event rates. For the random generation, we used a different probability distribution for each event, depending on the formula. For example, for the formula $\Diamond_{[0,5]}\, p$, the probability of p occurring was very small. All our logs consist of 100 different time-stamps, with the number of time-points labeled with the same time-stamp ranging from 100 to 100 000 on average per log. Overall, the log files comprised 8 GB of data. Their generation required more time than the actual monitoring task (at least for the LOCAL and GLOBAL version of our algorithms). GNU Parallel [18] was invaluable for both generating the logs and running the four tools on them.

Figure 5 shows our evaluation results. Each data point in the graphs represents the average of the maximum memory consumption over 10 randomly generated logs of a fixed average event rate. (The standard deviation is omitted in the figure as it was far below 1 MB for most time-points.) For all formulas, the space consumption of both the NAIVE version of our tool and MONPOLY increases linearly in the event rate, while for LOCAL and GLOBAL it stays almost constant. This relationship between the memory usage and the average event rate is consistent with our theoretical analysis. Moreover,

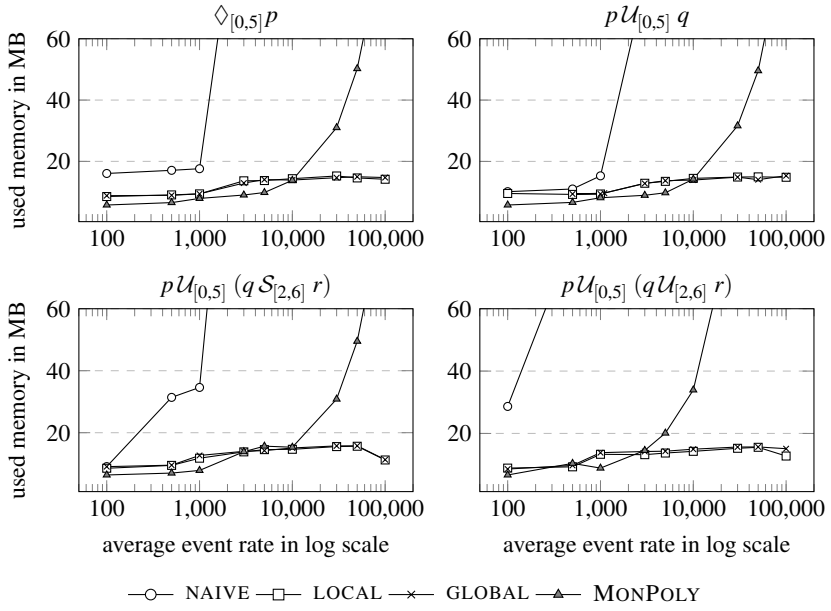

Fig. 5. Results of the experimental evaluation

LOCAL and GLOBAL do not differ essentially in memory consumption. We therefore advise using the LOCAL version of the algorithm given its additional guarantee of outputting equivalence verdicts only for time-points labeled with the same time-stamp.

Although we were not measuring time, increasing the memory consumption to 60 MB results in a significant increase in processing time per event, which leads to a much lower throughput for monitors like NAIVE and MONPOLY. This is not the case for our almost event-rate independent monitors.

7 Conclusion

We introduced the notion event-rate independence for measuring the space complexity of monitoring algorithms. This notion is desirable for monitors processing event streams of varying velocity. We presented a novel algorithm for monitoring metric temporal logic with bounded future operators that is almost event-rate independent. Our algorithm is concise and efficient.

As future work, we plan to study which extensions of metric temporal logic permit almost event-rate independent algorithms. Moreover, we intend to parallelize our algorithm, using existing frameworks in the spirit of Spark [21], to obtain monitors for expressive temporal logics that scale to big data applications.

Acknowledgment. Jasmin Blanchette, Srdjan Krstic, and anonymous TACAS reviewers helped to improve the presentation of this work. Bhatt is supported by the Swiss National Science Foundation grant Big Data Monitoring (167162).

References

[1] Aerial: An almost event-rate independent monitor for metric temporal logic. https://bitbucket.org/traytel/aerial (2016)

[2] Basin, D.A., Klaedtke, F., Müller, S., Pfitzmann, B.: Runtime monitoring of metric first-order temporal properties. In: FSTTCS 2008. pp. 49–60 (2008)

[3] Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. J. ACM 62(2), 15 (2015)

[4] Basin, D.A., Klaedtke, F., Zalinescu, E.: Algorithms for monitoring real-time properties. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 260–275. Springer (2011)

[5] Bauer, A., Küster, J., Vegliach, G.: From propositional to first-order monitoring. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 59–75. Springer (2013)

[6] Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. J. Log. Comput. 20(3), 651–674 (2010)

[7] Du, X., Liu, Y., Tiu, A.: Trace-length independent runtime monitoring of quantitative policies in LTL. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 231–247. Springer (2015)

[8] Furia, C.A., Spoletini, P.: Bounded variability of metric temporal logic. In: Cesta, A., Combi, C., Laroussinie, F. (eds.) TIME 2014. pp. 155–163. IEEE Computer Society (2014)

[9] Gunadi, H., Tiu, A.: Efficient runtime monitoring with metric temporal logic: A case study in the Android operating system. In: Jones, C.B., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 296–311. Springer (2014)

[10] Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: Katoen, J., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer (2002)

[11] Ho, H., Ouaknine, J., Worrell, J.: Online monitoring of metric temporal logic. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 178–192. Springer (2014)

[12] Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. 2(4), 255–299 (1990)

[13] Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. 78(5), 293–303 (2009)

[14] Maler, O., Nickovic, D., Pnueli, A.: Real time temporal logic: Past, present, future. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 2–16. Springer (2005)

[15] McAfee, A., Brynjolfsson, E.: Big data: The management revolution. Harvard Business Review 90(10), 61–67 (2012)

[16] Nipkow, T.: Boolean expression checkers. Archive of Formal Proofs (2014), http://isa-afp.org/entries/Boolean_Expression_Checkers.shtml

[17] Roşu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Automated Software Engineering 12(2), 151–197 (2005)

[18] Tange, O.: Gnu parallel - the command-line power tool. ;login: The USENIX Magazine 36(1), 42–47 (Feb 2011), http://www.gnu.org/s/parallel

[19] Thati, P., Roşu, G.: Monitoring algorithms for metric temporal logic specifications. Electr. Notes Theor. Comput. Sci. 113, 145–162 (2005)

[20] Ulus, D., Ferrère, T., Asarin, E., Maler, O.: Online timed pattern matching using derivatives. In: Chechik, M., Raskin, J.F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 736–751. Springer (2016)

[21] Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: Nahum, E.M., Xu, D. (eds.) HotCloud'10. USENIX Association (2010)