Noname manuscript No.
(will be inserted by the editor)

Almost Event-Rate Independent Monitoring

David Basin · Bhargav Nagaraja Bhatt · Srđan Krstić · Dmitriy Traytel

Received: date / Accepted: date

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 their 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, i.e., the number of events in a fixed time unit. This property is critical for monitoring voluminous streams of events with a high arrival rate.

We propose a new algorithm for metric temporal logic (MTL) that 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. Afterwards, we investigate more expressive logics. In particular, we extend linear dynamic logic with past operators and metric features. The resulting metric dynamic logic (MDL) offers the quantitative temporal conveniences of MTL while increasing its expressiveness. We show how to modify our MTL algorithm in a modular way, yielding an almost event-rate independent monitor for MDL. Finally, we compare our algorithms with traditional monitoring approaches, providing empirical evidence that almost event-rate independence matters in practice.

Keywords Runtime Verification · Monitoring · Temporal Logic · Regular Expressions

1 Introduction

Rules are integral to society. Companies and public institutions are often 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 automated monitoring tools can be used to prove compliance or identify violations.

In this paper we address the *online monitoring problem*: Given an unbounded stream of events and a property expressed in a formal specification language, identify all the points in the stream that violate the property. Monitoring algorithms come in two flavors: *online* algorithms, which analyze events as they occur in an unbounded stream, and *offline* algorithms,

D. Basin · B.N. Bhatt · S. Krstić · D. Traytel
Institute of Information Security, Departement of Computer Science, ETH Zürich, Zurich, Switzerland
E-mail: {basin,bbhatt,srdan.krstic,traytel}@inf.ethz.ch

which can analyze events stored in finite traces in any order. Compared with other verification problems, monitoring 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 consumption 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 consumption. The notion of *trace-length independence* requires a monitor's space complexity to be constant in the overall number of events [11]. Trace-length independence distinguishes monitors that can handle huge volumes of data from those that cannot. The classic 3V characterization by volume, velocity, and variety [33], tells us, however, that volume is only one challenging aspect of big data. Here, we account for another aspect: velocity or event rate.

Our first contribution is a new notion, *event-rate independence*, which states that a monitor's space consumption does not depend on the event rate, i.e., the number of events in a fixed time unit. We survey existing monitors (Section 2) for past-only linear temporal logic (ptLTL) [26] and its extension with metric intervals (ptMTL) [37] and we identify those monitors that have this property. For future-time operators, no such algorithms exists.

From a traditional standpoint, event-rate independent monitors for properties depending on future events seem impossible: these dependencies require the monitor to wait before it can output a *Boolean verdict* on whether the property holds. Because of this, traditional monitors only handle bounded future operators. But even in the bounded case, the sheer number of events that the monitor may need to wait for can be 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, whereby the monitor can store indices or pointers.

As our second contribution, we present almost event-rate independent monitoring algorithms for specification languages with past and future operators. We first focus on *metric temporal logic* (*MTL*) [30] interpreted over streams of time-stamped events (Section 3). This discrete semantics is based on integer *time-stamps*, which reflects the imprecision of physical clocks and is algorithmically easier to handle than a dense, interval-based model [10]. A finite number of consecutive events, each defining a *time-point*, might, however, carry the same time-stamp. The event rate is formally defined as the number of time-points per time-stamp (Section 4). 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.

To achieve almost event-rate independence, our monitor produces a different kind of output than traditional monitors (Section 5). Namely, it 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 may 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.

In reasoning about our monitor's space requirements, 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. Note that 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, months, or even years, for which the number of events fitting into one time unit increases dramatically. Beyond this logarithmic dependency, our monitor's space usage is independent of the event rate. This applies even to unbounded future operators, which our monitor handles without running out of memory.

Although our monitor's output is nonstandard, we are convinced of its usefulness. First, the output contains sufficient information to reconstruct all Boolean verdicts. Second, a monitor's users are often 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 unresolved time-point stays in the monitor's memory and is eventually output with a Boolean verdict. Thus, users will always see a truth value at the earliest violating event.

Our third contribution is to extend the specification language's expressiveness while retaining almost event-rate independence. In particular, LTL falls short of expressing all regular languages. For example, one cannot express that some event occurs at every other position in a stream. This lack of expressiveness is problematic in practice [41] and carries over to the point-based semantics of MTL [12]. A realistic example that cannot be expressed in MTL is that, within the next day, an action is approved and executed and the approval event happens before the execution event.

To overcome this limitation, researchers have developed numerous, more expressive extensions of LTL and MTL by extending these languages with regular-expression-like constructs [42]. This resulted in specification languages like the industrially standardized property specification language (PSL) [41], regular linear temporal logic (RLTL) [31, 35], and linear dynamic logic (LDL) [17]. We survey these and other languages (Section 2). All of them lack some of the features that make MTL an attractive choice: either its support for past operators or its quantitative features. We propose metric dynamic logic (MDL), an extension of LDL with past operators and quantitative features (Section 6) and extend our almost event-rate independent MTL algorithm to handle this more expressive language (Section 7). Our extension relies on Antimirov's partial derivatives of regular expressions [1].

We report on efficient implementations of our MTL and MDL monitoring algorithms (Section 8) and experimentally evaluate them, demonstrating that they outperform state-of-the-art monitoring tools for MTL and timed regular expressions (Section 9).

Taken together, our contributions lay the foundations for online monitoring that scales both with respect to the volume and the velocity of the event stream. This article is based on our earlier TACAS [3] and RV [7] conference papers. The additional contributions of this article are: (1) a unified presentation of the two works; (2) a simplified syntax for metric dynamic logic (Section 6); (3) more detailed correctness proofs of the algorithms (Subsections 5.3 and 7.3); (4) additional implementation details (Section 8); and (5) a substantially more extensive experimental evaluation (Section 9).

2 Related Work

We first survey existing trace-length and event-rate independent online monitoring algorithms for specification languages with a point-based time semantics and discrete time-stamps. Note that there are numerous monitoring algorithms for other time domains and semantics and we leave the study of event-rate independence in these settings as future work. Afterwards, we survey languages that can express ω-regular languages and associated monitoring algorithms.

Space Efficient Algorithms

We start with Havelund and Roşu [26] who propose a simple, yet efficient, online monitor for past-time linear temporal logic (ptLTL) using dynamic programming. The satisfaction relation of ptLTL at a given time-point in an event stream can be recursively defined in terms of the truth-values of subformulas at the previous time-point in the event stream. They exploit this to develop an algorithm that stores the truth-values of all subformulas at just the two latest time-points. Their algorithm's space complexity is $\mathcal{O}(n)$, where n is the formula's size.

Thati and Roşu [37] extend Havelund and Roşu's results to provide a trace-length independent, dynamic programming monitoring algorithm for MTL. The metric extension is handled by additionally storing the truth-values of all the interval-skewed subformulas. These are essentially the variants of the temporal subformulas that have their intervals skewed (or shifted) down by some constant. Note that this algorithm crucially relies on the values of time-stamps being discrete, otherwise the number of skewed subformulas would be infinite. The monitor's space complexity, therefore, depends only on the size of the formula and the constants occurring in its intervals. Hence, their monitor is event-rate independent. However, with no additional bookkeeping for the (unresolved) verdicts that depend on the future events, the algorithm essentially implements a non-standard semantics for MTL, truncated to finite traces. Namely, it outputs a verdict at each time-point without considering future events that can potentially 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 upon these dynamic programming techniques [26, 37]. However it implements the standard non-truncated semantics for MTL.

Basin et al. [4, 9] introduce techniques to handle MTL and metric first-order temporal logic with bounded future operators, adhering to the standard non-truncated semantics. 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 while it waits. The monitor's space complexity therefore grows linearly with the event rate, as is confirmed by their empirical evaluation [9, Section 6.3].

Trace-length independent (but not event-rate independent) monitoring algorithms have also been proposed for other temporal specification languages. Maler et al. [32] 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. There exist trace-length independent monitors for ptLTL extended with counting quantifiers [18], and ptMTL extended with recursive definitions [25].

Temporal Logic and ω-regular Languages

Dynamic LTL [27] (DLTL) is an early extension of LTL able to express all ω-regular languages. Leucker and Sánchez [31] propose regular LTL (RLTL), which improves upon DLTL by allowing regular expressions to be nested arbitrarily as LTL subformulas. RLTL's power operator is also more suitable for extensions that can handle the past. Sánchez and Leucker [35] extend RLTL with past operators and show that it can be translated into a 2-way alternating parity automaton with size linear in the size of the RLTL formula. But 2-way automata are not ideally suited for the online monitoring of high-velocity event streams, and removing bidirectionality incurs an exponential blowup [29]. Dax et al. [14] propose a similar extension of the property specification language [41] (PSL) with additional past operators,

called regular temporal logic (RTL). They translate RTL formulas into nondeterministic Büchi automata whose worst-case size is doubly exponential in the size of the RTL formula.

More recently, De Giacomo and Vardi [17] revisited this problem for the finite-trace semantics and introduced linear dynamic logic (LDL_f). This logic was inspired by the well-known propositional dynamic logic (PDL) [23], but its semantics closely resembles LTL. The authors do not discuss the past operators and do not provide a monitoring algorithm. However, they do provide a general translation from LDL formulas to alternating finite state automata that employs partial derivatives of regular expressions [1]. De Giacomo et al. [15, 16] provide a direct translation from LDL_f formulas to nondeterministic automata that are more suitable for monitoring. Faymonville et al. [20, 21] propose an extension of LDL called parametric linear dynamic logic (PLDL) that can specify quantitative temporal constraints. However, PLDL does not support past operators and its point-based time model does not include time-stamps, Time is instead implicitly encoded in the time-points.

Asarin et al. [2] introduce *timed regular expressions* (TRE) and prove their equivalence to timed automata. Additionally, the authors propose offline [39] and online [40] pattern matching algorithms for TRE, implemented as an open source tool called MONTRE [38]. Although TRE was originally defined over both discrete point-based and dense interval-based time models, MONTRE assumes the latter model. This makes it hard to use for monitoring streams with high event-rates in practice as the dense time model assumes that events can be sampled with arbitrary time granularity.

3 Metric Temporal Logic (MTL)

Metric temporal logic (*MTL*) [30] is a logic for specifying qualitative and quantitative temporal properties. We briefly describe MTL's syntax and point-based semantics over a discrete time domain. An in-depth discussion of various flavors of MTL is given elsewhere [10].

Let Σ be a set of atomic propositions. An *event* is a pair (τ, π), where $\tau \in \mathbb{N}$ is a *time-stamp* and $\pi \subseteq \Sigma$ is a set of propositions that are true at that event. An *event stream* is an infinite sequence of events $\rho = \langle(\tau_0, \pi_0), (\tau_1, \pi_1), (\tau_2, \pi_2), \ldots\rangle$, written $\langle(\pi_i, \tau_i)\rangle_{i\in\mathbb{N}}$, with monotonically increasing time-stamps: $\tau_i \leq \tau_{i+1}$ for all $i \in \mathbb{N}$. We write Δ_i for the non-negative time-stamp difference $\tau_{i+1} - \tau_i$. We call the indices in ρ *time-points*, i.e., the event (τ_i, π_i) occurs at time-point i. Moreover, we require that time progresses: for all time-stamps τ there exists a time-point i with $\tau_i > \tau$. These requirements allow successive time-points to have identical time-stamps; e.g., $\tau_0 = \tau_1 = \tau_2 = 5$. Hence, time-stamps may stutter, but only for finitely many time-points.

MTL's syntax is given by the grammar

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

where $p \in \Sigma$ and $I \in \mathbb{I}$. Here, $\mathbb{I}$ denotes the set of non-empty intervals over $\mathbb{N}$. We write $[a, b]$ for the interval $\{x \in \mathbb{N} \mid a \leq x \leq b\}$, where $a \in \mathbb{N}$, $b \in \mathbb{N} \cup \{\infty\}$, and $a \leq b$. For an interval I and $n \in \mathbb{N}$, we define $I - n$ to be the interval $\{x - n \mid x \in I\} \cap \mathbb{N}$ and I^- to be the set of intervals $\{I - n \mid n \in \mathbb{N}\}$, which is always finite. For instance, $[2,\infty)^-$ is $\{[2,\infty), [1,\infty), [0,\infty)\}$.

Along with the standard Boolean operators, MTL includes the past temporal operators $\bullet_I$ (*previous*) and $\mathcal{S}_I$ (*since*) and the future temporal operators $\bigcirc_I$ (*next*) and $\mathcal{U}_I$ (*until*), which may be nested freely. We omit the subscript I if $I = [0,\infty)$, and we use the usual syntactic sugar for the 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 the 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* $\blacksquare_I$.

A formula is interpreted with respect to a fixed event stream $\rho = \langle(\pi_i, \tau_i)\rangle_{i\in\mathbb{N}}$ at a time-point i. The dependence on the fixed ρ is left implicit in the following definition (also in many forthcoming ones), i.e., we write $i \models \varphi$ instead of the more standard $\rho, i \models \varphi$.

$$
\begin{array}{lll}
i \models p & \text{iff} & p \in \pi_i \\
i \models \neg\varphi & \text{iff} & i \not\models \varphi \\
i \models \varphi \vee \psi & \text{iff} & i \models \varphi \text{ or } i \models \psi \\
i \models \bullet_I \varphi & \text{iff} & i > 0 \text{ and } \Delta_{i-1} \in I \text{ and } i-1 \models \varphi \\
i \models \bigcirc_I \varphi & \text{iff} & \Delta_i \in I \text{ and } i+1 \models \varphi \\
i \models \varphi\, \mathcal{S}_I\, \psi & \text{iff} & j \models \psi \text{ for some } j \le i \text{ with } \tau_i - \tau_j \in I \text{ and } k \models \varphi \text{ for all } j < k \le i \\
i \models \varphi\, \mathcal{U}_I\, \psi & \text{iff} & j \models \psi \text{ for some } j \ge i \text{ with } \tau_j - \tau_i \in I \text{ and } k \models \varphi \text{ for all } i \le k < j
\end{array}
$$

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 as

$$i \models \varphi\, \mathcal{S}_I\, \psi \text{ iff } (a = 0 \wedge i \models \psi) \vee (i > 0 \wedge \Delta_{i-1} \le b \wedge i \models \varphi \wedge i-1 \models \varphi\, \mathcal{S}_{I-\Delta_{i-1}}\, \psi) \quad (1)$$

$$i \models \varphi\, \mathcal{U}_I\, \psi \text{ iff } (a = 0 \wedge i \models \psi) \vee (\Delta_i \le b \wedge i \models \varphi \wedge i+1 \models \varphi\, \mathcal{U}_{I-\Delta_i}\, \psi), \quad (2)$$

where $I = [a, b]$. 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 $\varphi \in \mathsf{SF}(\varphi)$. We say that ψ is a proper subformula of φ if $\psi \in \mathsf{SF}(\varphi) \setminus \{\varphi\}$. The set of interval-skewed subformulas of φ is defined as

$$
\begin{aligned}
\mathsf{ISF}(\varphi) = \mathsf{SF}(\varphi) \cup \{&\varphi_1\, \mathcal{S}_J\, \varphi_2 \mid \varphi_1\, \mathcal{S}_I\, \varphi_2 \in \mathsf{SF}(\varphi) \text{ and } J \in I^-\} \\
\cup \{&\varphi_1\, \mathcal{U}_J\, \varphi_2 \mid \varphi_1\, \mathcal{U}_I\, \varphi_2 \in \mathsf{SF}(\varphi) \text{ and } J \in I^-\}.
\end{aligned}
$$

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 is a proper subformula of ψ, then $\varphi < \psi$
- if $\varphi = \varphi_1\, \mathcal{S}_I\, \varphi_2$ and $\psi = \varphi_1\, \mathcal{S}_J\, \varphi_2$ and $J = I - n$ for some $n > 0$, then $\psi < \varphi$.

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. [28], which we subsequently use to analyze the complexity of our proposed algorithm.

$$
\begin{array}{lll}
\mathsf{FR}(p) = 0 & \mathsf{FR}(\neg\varphi) = \mathsf{FR}(\varphi) & \mathsf{FR}(\varphi \vee \psi) = \mathsf{max}(\mathsf{FR}(\varphi), \mathsf{FR}(\psi)) \\
\mathsf{FR}(\bullet_I \varphi) = \mathsf{FR}(\varphi) - \mathsf{inf}(I) & & \mathsf{FR}(\bigcirc_I \varphi) = \mathsf{sup}(I) + \mathsf{FR}(\varphi) \\
\mathsf{FR}(\varphi\, \mathcal{S}_I\, \psi) = \mathsf{max}(\mathsf{FR}(\varphi), \mathsf{FR}(\psi) - \mathsf{inf}(I)) & & \mathsf{FR}(\varphi\, \mathcal{U}_I\, \psi) = \mathsf{sup}(I) + \mathsf{max}(\mathsf{FR}(\varphi), \mathsf{FR}(\psi))
\end{array}
$$

Here max 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, i.e. a formula for which all subformulas of the form $\alpha\, \mathcal{U}_{[a,b]}\, \beta$, where $b \neq \infty$, 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 the 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	...
τ_i (time-stamps)	1	2	2	3	4	...
π_i (events)	$\{a\}$	$\{a\}$	$\{a\}$	$\{b\}$	$\{a,b\}$	...
$i \models a\,\mathcal{U}_{[0,1]}\,b$	$\bot$	$\top$	$\top$	$\top$	$\top$	...

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, even an algorithm with a space complexity exponential in the formula size is tolerable, but a space complexity linear in the trace length is problematic since this corresponds to storing the entire trace. Researchers have recently studied *trace-length independence* [11]. A monitor is trace-length independent if its efficiency does not decline as the number of events increases. In our setting, 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 exhaust 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 a specification language 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 [26, 37] 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 some specification language. 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 [24, 28, 32] 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 [9] 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 [9, 28], 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 the monitor 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 space requirement for an algorithm outputting such verdicts is therefore not event-rate independent but rather logarithmic in the event rate, since the size of such a block is bounded by 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 $\mathcal{M}$ 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 An Almost Event-Rate Independent Monitor for MTL

We first informally describe the high-level design of our MTL monitor. Afterwards we give a formal description and prove its correctness and almost event-rate independence.

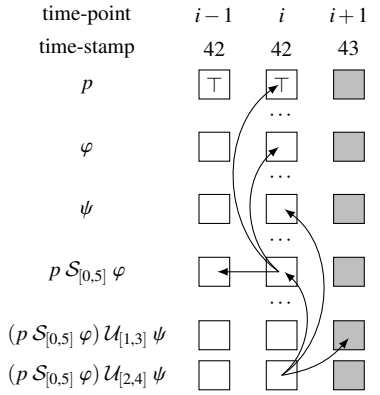
5.1 Informal Account

The idea of computing equivalence verdicts draws inspiration from the problem of simultaneous suffix matching with automata. To decide which suffixes of a word are matched by an automaton, a naive approach is to run the automaton starting 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 will travel through the automaton together. Our algorithm takes 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 [26], 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 [37] 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 in Havelund and Roşu's algorithm [26], but also supports future operators. The recursive Equation 2 for *until* reduces the satisfaction of a formula $\varphi \,\mathcal{U}_I\, \psi$ at the current time-point to a Boolean combination of the satisfaction of φ and ψ at the current time-point and the satisfaction of $\varphi \,\mathcal{U}_{I-n}\, \psi$ (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 Boolean 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. We may only output a verdict (i.e., the entry in the array indexed by the top-level formula) when its Boolean expression is resolved to a Boolean verdict. But until this happens, all such yet-to-be-output Boolean expressions must be separately stored, which affects the algorithm's space consumption.

To obtain almost event-rate independence, we refine our monitor by allowing it to output equivalence verdicts between different time-points. As soon as the monitor sees two semantically equivalent Boolean expressions for the toplevel formula, 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




```
now := −1
off := 0
hist := {}
prev := ⊥^n
curr := (Now ⊥)^n

step (τ, π) =
  Δ = τ − now
  for k = 0, …, n − 1
    prev[k] := eval_Δ(curr[k])
  hist := filter_verdicts
    ({((now, off), prev[Φ])} ∪
     {(j, subst(c)) | (j, c) ∈ hist})
  now := τ
  off := if Δ = 0 then off + 1 else 0
  for k = 0, …, n − 1
    curr[k] := progress (Φ_k, Δ, π)
```

Fig. 2: Example excerpt of the MTL monitor's state (left) and pseudocode (right)

only on the monitored formula φ. However, to enable users to understand equivalence verdicts, the equivalences must refer to different time-points represented using indices. Storing the indices requires logarithmic space in the event rate (as explained in Section 4). Hence, the overall algorithm is almost event-rate independent.

5.2 The Algorithm

At its core, our MTL monitor relies on the recursive Equations 1 and 2. The key observation is that the satisfiability of $\varphi\ \mathcal{U}_I\ \psi$ at time-point i is fully determined by the satisfiability of φ and ψ at the current time-point i and the satisfiability of the interval-skewed formula $\varphi\ \mathcal{U}_{I-\Delta_i}\ \psi$ at the next time-point $i+1$, along with some interval boundary checks. For $\mathcal{S}_I$, the symmetric characterization refers to an interval-skewed formula at the previous time-point $i-1$.

Figure 2 (left) illustrates these dependencies as arrows for verdicts at time point i for the formula $(p\ \mathcal{S}_{[0,5]}\ \varphi)\ \mathcal{U}_{[2,4]}\ \psi$ and its subformula $p\ \mathcal{S}_{[0,5]}\ \varphi$. Future dependencies can, in general, only be resolved after having seen the event at time-point $i+1$. Our monitor treats such future dependencies symbolically as Boolean variables. To monitor the formula Φ, the algorithm stores a Boolean expression for each interval-skewed subformula $\varphi \in \mathsf{ISF}(\Phi)$ in an array **curr**, ordered such that, for any formula φ at index k, each of its proper subformulas occurs at a lower index $l < k$. We write Φ_k for the formula occurring at index k and sometimes identify formulas with their indices, for example, by writing **curr**$[\varphi]$ for the **curr**'s entry at position k, given that $\varphi = \Phi_k$. We use Boolean expressions in negation normal form, defined as:

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

Negation $\neg$ is applied to arbitrary Boolean expressions by pushing it down to the leaves.

With each arriving event, the array **curr** is updated following the Equations 1 and 2 or using the MTL semantics directly. The variables in expressions at time-point i represent pointers into the monitor's array **curr** *after* processing the event at time-point $i+1$. Instead of using pointers to the past time-point $i-1$, the monitor directly uses the expressions from the array at time-point $i-1$ to build from them new expressions at time-point i.

For future formulas, there is an additional complication: before the monitor has seen the time-stamp at position $i+1$, it cannot know which of the interval-skewed future formulas to

$$\begin{aligned}
\mathsf{subst}(\bot) &= \bot & \mathsf{subst}_{fexp}(\bot) &= \mathsf{Now}\ \bot \\
\mathsf{subst}(\top) &= \top & \mathsf{subst}_{fexp}(\top) &= \mathsf{Now}\ \top \\
\mathsf{subst}(\mathsf{Var}\ k) &= \mathbf{prev}[k] & \mathsf{subst}_{fexp}(\mathsf{Var}\ k) &= \mathbf{curr}[k] \\
\mathsf{subst}(\neg \mathsf{Var}\ k) &= \neg(\mathbf{prev}[k]) & \mathsf{subst}_{fexp}(\neg \mathsf{Var}\ k) &= \neg_{fexp}(\mathbf{curr}[k]) \\
\mathsf{subst}(b \wedge c) &= \mathsf{subst}(b) \wedge \mathsf{subst}(c) & \mathsf{subst}_{fexp}(b \wedge c) &= \mathsf{subst}_{fexp}(b) \wedge_{fexp} \mathsf{subst}_{fexp}(c) \\
\mathsf{subst}(b \vee c) &= \mathsf{subst}(b) \vee \mathsf{subst}(c) & \mathsf{subst}_{fexp}(b \vee c) &= \mathsf{subst}_{fexp}(b) \vee_{fexp} \mathsf{subst}_{fexp}(c)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{filter_verdicts}\ H = H \setminus \\
&\quad (\{\mathsf{output_bool}((\tau,k),b) \mid ((\tau,k),b) \in H \wedge (b = \top \vee b = \bot)\} \cup \\
&\quad \{\mathsf{output_equiv}(((\tau,k),b),(\tau',l)) \mid \mathsf{mode}(\tau,\tau') \wedge \\
&\qquad ((\tau,k),b) \in H \wedge ((\tau',l),c) \in H \wedge b \equiv c \wedge (\tau',l) < (\tau,k)\})
\end{aligned}$$

$$\mathsf{mode}(\tau,\tau') = \begin{cases} \bot & \text{NAIVE} \\ \top & \text{GLOBAL} \\ \tau = \tau' & \text{LOCAL} \end{cases}$$

Fig. 3: Auxiliary functions

refer to. We therefore work with so called *future expressions* defined as $fexp = \mathsf{Now}\ bexp \mid \mathsf{Later}\ (\mathbb{N} \to bexp)$, where the parameter of Later is the time-stamp difference between the time-points $i+1$ and i. The functions $\wedge_{fexp}$, $\vee_{fexp}$, and $\neg_{fexp}$ lift Boolean operators to future expressions while propagating Boolean values eagerly, for example, by simplifying $\mathsf{Now}\ \top \vee_{fexp} x$ to $\mathsf{Now}\ \top$ and $\mathsf{Now}\ \bot \vee_{fexp} x$ to x. Given a time-stamp difference Δ, a future expression evaluates to a Boolean expression: $\mathsf{eval}_\Delta(\mathsf{Now}\ c) = c$ and $\mathsf{eval}_\Delta(\mathsf{Later}\ f) = f(\Delta)$.

We are now set to describe our monitor. Figure 2 (right) depicts its (OCaml-like) pseudocode. The monitor's state consists of five variables initialized as shown, where n is the number of interval-skewed subformulas of Φ. To denote their mutability, we write the variables in boldface. The variable **now** is the current time-stamp and together with it the natural number **off** identifies the current time-point. Note that we represent time-points as pairs (τ, k), where the second component is an offset into a block of time-points labeled with time-stamp τ. The history **hist** is a set of pairs of Boolean expressions and time-points (again stored as time-stamp-offset pairs). It contains all time-points at which no verdict was output so far, since the verdict depends on future events. The variable **curr** is the array of length n of future expressions for all interval-skewed subformulas at the current time-point. The variable **prev** is another array of length n of Boolean expressions that belong to the previous time-point. The monitor updates its state using the step function for each incoming event (τ, π).

The step function first computes the time-stamp difference Δ between τ and the previous time-stamp stored in **now**. It uses Δ to evaluate future expressions from **curr** to Boolean expressions and store them in **prev**, thereby discarding any old expression stored in **prev**. Next it updates the history **hist**. This step is the key to obtaining almost event-rate independence. The variables of all Boolean expressions stored in the history refer to the last seen time-point. To maintain this invariant, we first update all expressions in the history by substituting their variables (pointing to what used to be in **curr** before the call of step) with the actual Boolean expressions contained in **curr** (that is now stored in **prev**). The substitution is performed by the function subst shown in Figure 3 (top left). Moreover, the expression $\mathbf{prev}[\Phi]$ is added as a new element to the history. Then, the function filter_verdicts, shown in Figure 3 (bottom left), performs two verdict output steps. First, it iterates over the history and removes (using the identity function output_bool, which as a side-effect outputs a Boolean verdict) all Boolean expressions equivalent to $\top$ or $\bot$. Second, it finds all pairwise equivalent pairs of expressions from the history and for each such pair it removes the expression with the larger time-point from the history (using output_equiv which simply returns its first argument, and

as a side-effect outputs an equivalence verdict of the form $(\tau,k) \equiv (\tau',l)$). By guaranteeing that only semantically different Boolean expressions in at most n variables are contained in the history, the monitor is almost event-rate independent, as only the offset components of the time-points stored in history have sizes that depend on the event rate.

If the second step were omitted, we would obtain a standard monitor that is event-rate dependent. In fact, we employ a *mode* flag in the definition of filter_verdicts to switch between this event-rate dependent NAIVE mode of operation and the above almost event-rate independent mode, which we call GLOBAL. The third mode, LOCAL constitutes a middle ground between NAIVE and GLOBAL: it outputs equivalence verdicts only for time-points with the same time-stamp. This mode is still almost event-rate independent, while requiring fewer Boolean equivalence check to be performed. Finally, after a trivial update of **now** and **off**, the progress function fills the **curr** array with new future expressions.

$$
\begin{aligned}
&\mathsf{progress}\ (\varphi, \Delta, \pi) = \mathsf{case}\ \varphi\ \mathsf{of}\\
&\quad \mid p \qquad\qquad\ \Rightarrow \mathsf{Now}\ (p \in \pi)\\
&\quad \mid \neg\psi \qquad\qquad \Rightarrow \neg_{fexp}(\mathbf{curr}[\psi])\\
&\quad \mid \psi_1 \vee \psi_2 \qquad \Rightarrow \mathbf{curr}[\psi_1] \vee_{fexp} \mathbf{curr}[\psi_2]\\
&\quad \mid \bullet_I \psi \qquad\quad \Rightarrow \mathsf{Now}\ (\Delta \in I) \wedge_{fexp} \mathsf{subst}_{fexp}(\mathbf{prev}[\psi])\\
&\quad \mid \bigcirc_I \psi \qquad\quad \Rightarrow \mathsf{Later}\ (\lambda\Delta'.\ \Delta' \in I \wedge \mathsf{Var}\ \psi)\\
&\quad \mid \psi_1\ \mathcal{S}_{[a,b]}\ \psi_2 \Rightarrow (\mathsf{Now}\ (a = 0) \wedge_{fexp} \mathbf{curr}[\psi_2]) \vee_{fexp}\\
&\qquad\qquad\qquad\quad (\mathsf{Now}\ (\Delta \leq b) \wedge_{fexp} \mathbf{curr}[\psi_1] \wedge_{fexp} \mathsf{subst}_{fexp}(\mathbf{prev}[\psi_1\ \mathcal{S}_{[a,b]-\Delta}\ \psi_2]))\\
&\quad \mid \psi_1\ \mathcal{U}_{[a,b]}\ \psi_2 \Rightarrow (\mathsf{Now}\ (a = 0) \wedge_{fexp} \mathbf{curr}[\psi_2]) \vee_{fexp}\\
&\qquad\qquad\qquad\quad \mathsf{Later}\ (\lambda\Delta'.\ \Delta' \leq b \wedge \mathsf{eval}_{\Delta'}(\mathbf{curr}[\psi_1]) \wedge \mathsf{Var}\ (\psi_1\ \mathcal{U}_{[a,b]-\Delta'}\ \psi_2))
\end{aligned}
$$

The first three cases of the definition are straightforward. After passing the metric condition check, the case $\bullet_I$ accesses the **prev** array to obtain the Boolean expression representing the satisfaction of ψ at the previous time-point. Note that the variables in this Boolean expression refer to the current time-point. The function subst_{fexp}, shown in Figure 3 (top right), is used to lift such Boolean expressions to future Boolean expressions that refer to the next time-point and fit precisely what should be stored in the **curr** array. For $\bigcirc_I$ the situation is similar, except that the parameter Δ is not used, as it is the time-stamp difference between the current and previous time-point. In contrast, Δ' from the Later argument is the appropriate time-stamp difference for $\bigcirc_I$. It will be instantiated to a concrete value after the next event (including its time-stamp) is received. To refer to the satisfiability of ψ at time-point $i+1$, we use the Var constructor applied to (the index of) ψ. For the last two cases, progress follows the recursive Equations 1 and 2 and behaves similarly to $\bullet_I$ and $\bigcirc_I$ when accessing the previous and next time-points. For example, to refer to the satisfiability of $\psi_1\ \mathcal{U}_{[a,b]-\Delta'}\ \psi_2$ at time-point $i+1$ as stipulated by Equation 2, we use the Var constructor applied to (the index of) $\psi_1\ \mathcal{U}_{[a,b]-\Delta'}\ \psi_2$.

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 three rows show the time-points, time-stamps, and the events from ρ. The next four rows show the values Δ, **hist**, **now**, and **off** in the order in which they are updated in the step function. The following 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 future expressions stored in **curr**, we show instead the result of evaluating the expressions in **curr** with the time-stamp difference Δ_i to the next time-point, i.e., $\mathsf{eval}_{\Delta_i}(\mathbf{curr}[-])$. This causes a delay of one time-point between the values in the arrays and the history updates and verdict outputs.

i		0	1	2	3	4
τ_i		1	2	2	3	4
π_i		$\{a\}$	$\{a\}$	$\{a\}$	$\{b\}$	$\{a,b\}$
Δ_{i-1}		2	1	0	1	1
hist	$\{\}$	$\{\}$	$\{((1,0), \mathsf{Var}\,\varphi_0)\}$	$\{((2,0), \mathsf{Var}\,\varphi_1), ((1,0), \mathsf{Var}\,\varphi_0)\}$	$\{((2,0), \mathsf{Var}\,\varphi_0)\}$	$\{\}$
now	-1	1	2	2	3	4
off	0	0	0	1	0	0
Table for $\mathsf{eval}_{\Delta_i}(\mathbf{curr}[-])$						
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.

$$
\begin{aligned}
\mathcal{I} = (\mathcal{I}1) \quad & (\forall((\tau,i),b) \in \mathbf{hist}.\ \tau@i \models \Phi \leftrightarrow \mathbf{now@off} \models_{bexp} b) \\
\wedge\ (\mathcal{I}2) \quad & (\forall\varphi \in \mathsf{ISF}(\Phi).\ \mathbf{now@off} \models \varphi \leftrightarrow (\mathbf{now@off}) + 1 \models_{bexp} \mathsf{eval}_{\Delta_{\mathbf{now@off}}}(\mathbf{curr}[\varphi]) \\
\wedge\ (\mathcal{I}3) \quad & (\forall\varphi \in \mathsf{ISF}(\Phi).\ (\mathbf{now@off}) - 1 \models \varphi \leftrightarrow \mathbf{now@off} \models_{bexp} \mathbf{prev}[\varphi] \\
\wedge\ (\mathcal{I}4) \quad & (\forall\varphi \in \mathsf{ISF}(\Phi).\ \mathsf{Vars}\,(\mathsf{eval}_{\Delta_{\mathbf{now@off}}}(\mathbf{curr}[\varphi])) \cup \mathsf{Vars}\,(\mathbf{prev}[\varphi]) \subseteq \mathsf{ISF}(\varphi)) \\
\wedge\ (\mathcal{I}5) \quad & (\forall((\tau,i),b) \in \mathbf{hist}.\ b \not\equiv \top \wedge b \not\equiv \bot) \\
\wedge\ (\mathcal{I}6) \quad & (\forall((\tau,i),b) \in \mathbf{hist}.\ \forall((\tau',j),c) \in \mathbf{hist}.\ \tau@i \neq \tau'@j \rightarrow \mathsf{compact}(\tau,\tau',b,c))
\end{aligned}
$$

We write $\tau@i$ to denote the time-point uniquely identified by the time-stamp τ and the offset i within the time-stamp. Moreover, Vars is the set of variables in a Boolean expression 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. $(\mathcal{I}1)$, $(\mathcal{I}2)$, and $(\mathcal{I}3)$ capture the semantics of the entries in the history and the expression arrays. $(\mathcal{I}4)$ expresses that future dependencies in any expression indexed by a subformula φ may only refer to φ's interval-skewed subformulas. $(\mathcal{I}5)$ guarantees that Boolean verdicts are output as early as possible. $(\mathcal{I}6)$ is crucial for our complexity analysis. It uses an auxiliary predicate $\mathsf{compact}$, defined differently for each of the three modes 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 for future subformulas. The **curr** 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 *$\mathcal{I}$ holds after processing the first event using* step. *Moreover, $\mathcal{I}$ is preserved by processing any event using* step.

Proof The key step is the preservation of $(\mathcal{I}2)$ by the progress function when processing the time-point $i > 0$. We prove the following auxiliary lemma: Fix a stream $\rho = \langle(\pi_i, \tau_i)\rangle_{i\in\mathbb{N}}$ and the formula Φ. Let $0 < i = \mathbf{now}@\mathbf{off}$, $\varphi \in \mathsf{ISF}(\Phi)$, and $fb = \mathsf{progress}(\varphi, \Delta_{i-1}, \pi_i)$. Assume that (i) for all $\varphi \in \mathsf{ISF}(\Phi)$ we have $i-1 \models \varphi \leftrightarrow i \models_{bexp} \mathbf{prev}[\varphi]$ and (ii) for all proper subformulas ψ of φ, we have $i \models \psi \leftrightarrow i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\mathbf{curr}[\varphi])$. Then $i \models \varphi \leftrightarrow i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(fb)$.

The lemma follows by a case distinction on φ:

$\varphi = p$: We have $fb = \mathsf{Now}\ (p \in \pi_i)$ and hence

$$i \models \varphi \leftrightarrow p \in \pi_i \leftrightarrow i+1 \models_{bexp} p \in \pi_i \leftrightarrow i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(fb).$$

$\varphi = \neg\psi$: We have $fb = \neg_{fexp}(\mathbf{curr}[\psi])$ and hence

$$\begin{aligned} i \models \varphi &\leftrightarrow i \not\models \psi \overset{\text{(ii)}}{\leftrightarrow} i+1 \not\models_{bexp} \mathsf{eval}_{\Delta_i}(\mathbf{curr}[\psi]) \leftrightarrow i+1 \models_{bexp} \neg(\mathsf{eval}_{\Delta_i}(\mathbf{curr}[\psi])) \\ &\leftrightarrow i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\neg_{fexp}(\mathbf{curr}[\psi])) \leftrightarrow i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(fb). \end{aligned}$$

$\varphi = \psi_1 \wedge \psi_2$: We have $fb = \mathbf{curr}[\psi_1] \wedge_{fexp} \mathbf{curr}[\psi_2]$ and hence

$$\begin{aligned} i \models \varphi &\leftrightarrow i \models \psi_1 \wedge i \models \psi_2 \overset{\text{twice (ii)}}{\leftrightarrow} i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\mathbf{curr}[\psi_1]) \wedge i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\mathbf{curr}[\psi_2]) \\ &\leftrightarrow i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\mathbf{curr}[\psi_1]) \wedge \mathsf{eval}_{\Delta_i}(\mathbf{curr}[\psi_2]) \\ &\leftrightarrow i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\mathbf{curr}[\psi_1] \wedge_{fexp} \mathbf{curr}[\psi_2]) \leftrightarrow i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(fb). \end{aligned}$$

$\varphi = \bullet_I \psi$: We have $fb = \mathsf{Now}\ (\Delta_{i-1} \in I) \wedge_{fexp} \mathsf{subst}_{fexp}(\mathbf{prev}[\psi])$ and hence

$$\begin{aligned} i \models \varphi &\overset{i>0}{\leftrightarrow} \Delta_{i-1} \in I \wedge i-1 \models \psi \overset{\text{(i)}}{\leftrightarrow} \Delta_{i-1} \in I \wedge i \models_{bexp} \mathbf{prev}[\psi] \\ &\overset{(*)}{\leftrightarrow} \Delta_{i-1} \in I \wedge i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\mathsf{subst}_{fexp}(\mathbf{prev}[\psi])) \\ &\leftrightarrow i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\mathsf{Now}\ (\Delta_{i-1} \in I) \wedge_{fexp} \mathsf{subst}_{fexp}(\mathbf{prev}[\psi])) \\ &\leftrightarrow i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(fb). \end{aligned}$$

Here, the step labeled with $(*)$ uses an auxiliary lemma about subst_{fexp}: for all Boolean expressions c we have $i \models_{bexp} c \leftrightarrow i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\mathsf{subst}_{fexp}(c))$, provided that $\mathsf{Vars}(c) \subseteq A$ and for all $\psi \in A$ we have $i \models \psi \leftrightarrow i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\mathbf{curr}[\varphi])$. We use this lemma with $A = \mathsf{ISF}(\psi)$ using $(\mathcal{I}6)$ and (ii) to discharge the lemma's assumptions.

$\varphi = \bigcirc_I \psi$: We have $fb = \mathsf{Later}\ (\lambda\Delta'.\, \Delta' \in I \wedge \mathsf{Var}\ \psi)$ and hence

$$\begin{aligned} i \models \varphi &\leftrightarrow \Delta_i \in I \wedge i+1 \models \psi \leftrightarrow \Delta_i \in I \wedge i+1 \models_{bexp} \mathsf{Var}\ \psi \\ &\leftrightarrow i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\mathsf{Later}\ (\lambda\Delta'.\, \Delta' \in I \wedge \mathsf{Var}\ \psi)) \leftrightarrow i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(fb). \end{aligned}$$

$\varphi = \psi_1\ \mathcal{S}_{[a,b]}\ \psi_2$: We have $fb = (\mathsf{Now}\ (a = 0) \wedge_{fexp} \mathbf{curr}[\psi_2]) \vee_{fexp}$ $(\mathsf{Now}\ (\Delta_{i-1} \le b) \wedge_{fexp} \mathbf{curr}[\psi_1] \wedge_{fexp} \mathsf{subst}_{fexp}(\mathbf{prev}[\psi_1\ \mathcal{S}_{[a,b]-\Delta_{i-1}}\ \psi_2]))$ and hence

$$\begin{aligned} i \models \varphi &\overset{\text{Eq. 1, } i>0}{\leftrightarrow} (a = 0 \wedge i \models \psi_2) \vee (\Delta_{i-1} \le b \wedge i \models \psi_1 \wedge i-1 \models \psi_1\ \mathcal{S}_{[a,b]-\Delta_{i-1}}\ \psi_2) \\ &\overset{\text{(i), (ii)}}{\leftrightarrow} (a = 0 \wedge i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\mathbf{curr}[\psi_2])) \vee \\ &\quad (\Delta_{i-1} \le b \wedge i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\mathbf{curr}[\psi_1]) \wedge i \models_{bexp} \mathbf{prev}[\psi_1\ \mathcal{S}_{[a,b]-\Delta_{i-1}}\ \psi_2]) \\ &\overset{(*)}{\leftrightarrow} i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(fb). \end{aligned}$$

Here, the step labeled with $(*)$ uses the same lemma about subst_{fexp} as in the $\bullet_I$ case.

$\varphi = \psi_1 \,\mathcal{U}_{[a,b]}\, \psi_2$: We have $fb = (\mathsf{Now}\ (a=0) \wedge_{fexp} \mathbf{curr}[\psi_2]) \vee_{fexp}$
$\mathsf{Later}\ (\lambda \Delta'.\ \Delta' \leq b \wedge \mathsf{eval}_{\Delta'}(\mathbf{curr}[\psi_1]) \wedge \mathsf{Var}\ (\psi_1 \,\mathcal{U}_{[a,b]-\Delta'}\, \psi_2))$ and hence

$$\begin{aligned} i \models \varphi &\overset{\text{Eq. 2}}{\leftrightarrow} (a=0 \wedge i \models \psi_2) \vee (\Delta_i \leq b \wedge i \models \psi_1 \wedge i+1 \models \psi_1 \,\mathcal{U}_{[a,b]-\Delta_i}\, \psi_2) \\ &\overset{\text{(ii)}}{\leftrightarrow} (a=0 \wedge i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\mathbf{curr}[\psi_2])) \vee \\ &\quad\ (\Delta_i \leq b \wedge i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\mathbf{curr}[\psi_1]) \wedge i+1 \models_{bexp} \mathsf{Var}\ (\psi_1 \,\mathcal{U}_{[a,b]-\Delta_i}\, \psi_2)) \\ &\leftrightarrow\ i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(fb). \end{aligned}$$

This concludes the proof of the auxiliary lemma. To conclude the overall proof of the preservation of $(\mathcal{I}2)$, the assumption (i) and (ii) of our auxiliary lemma are discharged using the invariants $(\mathcal{I}2)$ and $(\mathcal{I}3)$ for the previous state of the monitor. The other invariants are easy to establish by following the monitor's execution shown in Figure 2 (right). □

The step from the invariant to a correctness theorem is easy. For soundness, we calculate, using $(\mathcal{I}1)$, the expected semantic properties for verdicts that are output in a `step`. Completeness is more delicate: a violation of a liveness property, such as $\square(\Diamond p)$, cannot be detected by a monitor. However, for safety properties that can be expressed using bounded future formulas, we can guarantee that for every time-point a verdict is eventually output by our algorithm. This is easy to see: for each time-point either a verdict is output or the corresponding expression remains in the history. And 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 $((\tau, i), b)$ 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 on bounded future formulas.*

Finally, we establish complexity bounds. Let $n = |\mathsf{ISF}(\Phi)|$ and $d = \mathsf{FR}(\varphi)$. Note that $d \leq n$ for bounded future formulas. The size of a Boolean expression in n variables can be bounded by 2^n assuming a normal form for expressions such as DNF. Then the size of the future-dependent array **curr** 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 **curr**.

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. □

The above bounds apply to arbitrary formulas. For liveness properties, such as $\square(\Diamond p)$, the NAIVE and LOCAL bounds is useless, since $d = \infty$. In fact, the NAIVE and LOCAL algorithms will eventually run out of memory for this formula. Interestingly, the GLOBAL algorithm can handle such liveness properties gracefully in terms of memory consumption. Of course, they will still never output Boolean verdicts for such formulas, but only equivalence verdicts.

To process an event, our monitor solves several NP-complete problem instances. However, the Boolean equivalences arising in practice are simple and tractable (Section 9).

6 Metric Dynamic Logic (MDL)

In this section, we introduce metric dynamic logic (MDL). This logic extends LDL [17] with past temporal operators and time intervals associated with temporal formulas. Moreover, we further syntactically simplify LDL's temporal operators without losing expressiveness.

MDL's syntax is defined by the following grammar, where $p \in \Sigma$ denotes an atomic proposition, and $I \in \mathbb{I}$ denotes a non-empty interval.

$$\psi = p \mid \neg\psi \mid \psi \vee \psi \mid |r\rangle_I \mid \langle r|_I \qquad\qquad r = \star \mid \psi? \mid r + r \mid r \cdot r \mid r^*$$

Aside from Boolean operators, MDL contains the regular expressions modalities like the metric future match operator $|r\rangle_I$, which expresses that there exists some future time-point with a time difference bounded by the interval I and the regular expression r matches the portion of the event stream from the current point up to that future time-point. Notably, the regular expression itself may nest arbitrary MDL formulas. The past match operator $\langle r|_I$ expresses the same property about a past time-point. Regular expressions in MDL match portions of the event stream, i.e., words over 2^Σ. The expression $\star$ matches any character and $\varphi?$ matches the empty word starting at time-point i if the formula φ holds at i. Moreover, $+$, $\cdot$, and * are the standard sum, concatenation, and (Kleene) star operators.

The semantics of formulas and regular expressions are defined by mutual induction. A formula is interpreted over a fixed event stream $\rho = \langle(\tau_i, \pi_i)\rangle_{i\in\mathbb{N}}$ and a position $i \in \mathbb{N}$. The semantics of a regular expression r is given by a relation $\mathcal{R}(r) \subseteq \mathbb{N} \times \mathbb{N}$ that contains pairs of time-points (i, j) with $i \le j$ (or $j \le i$ for past) such that the sequence $\pi_i, \ldots, \pi_j$ (or $\pi_j, \ldots, \pi_i$ for past) from the fixed ρ matches r.

$$\begin{aligned}
i &\models p \text{ iff } p \in \pi_i & \mathcal{R}(\star) &= \{(i, i+1) \mid i \in \mathbb{N}\}\\
i &\models \neg\varphi \text{ iff } i \not\models \varphi & \mathcal{R}(\varphi?) &= \{(i, i) \mid i \models \varphi\}\\
i &\models \varphi \vee \psi \text{ iff } i \models \varphi \text{ or } i \models \psi & \mathcal{R}(r+s) &= \mathcal{R}(r) \cup \mathcal{R}(s)\\
i &\models |r\rangle_I \text{ iff there exists } j \ge i & \mathcal{R}(r \cdot s) &= \{(i,k) \mid \exists j.\ (i,j) \in \mathcal{R}(r) \wedge (j,k) \in \mathcal{R}(s)\}\\
&\quad \text{with } \tau_j - \tau_i \in I \text{ and } (i,j) \in \mathcal{R}(r) & \mathcal{R}(r^*) &= \{(i,i) \mid i \in \mathbb{N}\} \cup\\
i &\models \langle r|_I \text{ iff there exists } j \le i & &\quad \{(i_0, i_k) \mid \exists i_1, \ldots, i_{k-1}.\ (i_j, i_{j+1}) \in \mathcal{R}(r)\\
&\quad \text{with } \tau_i - \tau_j \in I \text{ and } (j,i) \in \mathcal{R}(r) & &\qquad \text{for all } 0 \le j < k\}
\end{aligned}$$

We employ the usual syntactic sugar for additional Boolean constants and operators: $\mathit{true} = p \vee \neg p$, $\mathit{false} = \neg\mathit{true}$, and $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$. The future diamond dynamic modality of LDL [17], which we had previously extended with metric intervals and the dual past operator given in our earlier formulation of MDL [7], can be easily defined in terms of the match operators: $\langle r\rangle_I\, \varphi = |r \cdot \varphi?\rangle_I$ and $\varphi\, {}_I\langle r\rangle = \langle \varphi? \cdot r|_I$. The ability to arbitrarily nest the negation operator allows us to define the metric future and past box operators $[r]_I\, \varphi$ and $\varphi\, {}_I[r]$ as $\neg(\langle r\rangle_I\, \neg\varphi)$ and $\neg(\neg\varphi\, {}_I\langle r\rangle)$, respectively. We use the abbreviations $|\varphi\rangle_I$, $\langle\varphi|_I$, $\langle\varphi\rangle_I\, \psi$, and $\psi\, {}_I\langle\varphi\rangle$ for $|\varphi? \cdot \star\rangle_I$, $\langle \star \cdot \varphi?|_I$, $\langle \varphi? \cdot \star\rangle_I\, \psi$, and $\psi\, {}_I\langle \star \cdot \varphi?\rangle$, respectively. We perform the same implicit *cast* of a formula φ to the regular expression $\varphi? \cdot \star$ in the context of a future regular expression (or $\star \cdot \varphi?$ in the context of a past regular expression) for any formula that occurs as an argument to one of the $+$, $\cdot$, and * constructors. For example, $\langle \varphi^*\rangle_I\, \psi$ abbreviates $\langle (\varphi? \cdot \star)^*\rangle_I\, \psi$.

For an MDL formula φ, let $\mathsf{SF}(\varphi)$ denote the set of its subformulas defined as usual. (We overload this and other notions, which are used for both MTL and MDL.) Note that $\varphi \in \mathsf{SF}(\varphi)$. We say that ψ is a proper subformula of φ if $\psi \in \mathsf{SF}(\varphi) \setminus \{\varphi\}$. We define the set of *interval-skewed subformulas* $\mathsf{ISF}(\varphi)$ as $\mathsf{SF}(\varphi) \cup \{\langle r|_J \mid \langle r|_I \in \mathsf{SF}(\varphi), J \in I^-\} \cup \{|r\rangle_J \mid |r\rangle_I \in \mathsf{SF}(\varphi), J \in I^-\}$, which contains all temporal formulas with the same structure as existing temporal subformulas of φ, except with intervals shifted by constants.

Theorem 3 *For every MTL formula there exists an equivalent MDL formula.*

Proof We prove this constructively by defining a syntactic translation ξ:

$$\xi(p) = p; \quad \xi(\varphi \vee \psi) = \xi(\varphi) \vee \xi(\psi); \quad \xi(\neg\varphi) = \neg\xi(\varphi); \quad \xi(\bigcirc_I \varphi) = \langle\star\rangle_I\, \xi(\varphi);$$
$$\xi(\varphi\, \mathcal{U}_I\, \psi) = \langle \xi(\varphi)^*\rangle_I\, \xi(\psi); \quad \xi(\bullet_I\, \varphi) = \xi(\varphi)\,{}_I\langle\star\rangle; \quad \xi(\varphi\, \mathcal{S}_I\, \psi) = \xi(\psi)\,{}_I\langle \xi(\varphi)^*\rangle.$$

Given an MTL formula φ and a fixed stream ρ, we prove that $\forall i.\ i \models \xi(\varphi) \leftrightarrow i \models \varphi$ by induction on the structure of φ. (Note that we overload the notation for satisfiability $\models$ for both logics.) We show the proof only for $\mathcal{U}_I$. The other cases follow similarly.

$$\begin{aligned}
& i \models \xi(\varphi\, \mathcal{U}_I\, \psi) \overset{\text{def. } \xi}{\longleftrightarrow} i \models \langle \xi(\varphi)^*\rangle_I\, \xi(\psi) \overset{\text{desugar}}{\longleftrightarrow} i \models |(\xi(\varphi)? \cdot \star)^* \cdot \xi(\psi)?\rangle_I \\
& \overset{\text{def. } \models}{\longleftrightarrow} \text{ there exists } j \geq i \text{ with } \tau_j - \tau_i \in I \text{ and } (i,j) \in \mathcal{R}((\xi(\varphi)? \cdot \star)^* \cdot \xi(\psi)?) \\
& \overset{\text{def. } \mathcal{R}}{\longleftrightarrow} j \models \xi(\psi) \text{ for some } j \geq i \text{ with } \tau_j - \tau_i \in I \text{ and } (i,j) \in \mathcal{R}((\xi(\varphi)? \cdot \star)^*) \\
& \overset{\text{IH } \psi}{\longleftrightarrow} j \models \psi \text{ for some } j \geq i \text{ with } \tau_j - \tau_i \in I \text{ and } (i,j) \in \mathcal{R}((\xi(\varphi)? \cdot \star)^*) \\
& \overset{\text{def. } \mathcal{R}}{\longleftrightarrow} j \models \psi \text{ for some } j \geq i \text{ with } \tau_j - \tau_i \in I \text{ and } k \models \xi(\varphi) \text{ for all } i \leq k < j \\
& \overset{\text{IH } \varphi}{\longleftrightarrow} j \models \psi \text{ for some } j \geq i \text{ with } \tau_j - \tau_i \in I \text{ and } k \models \varphi \text{ for all } i \leq k < j \\
& \overset{\text{def. } \models}{\longleftrightarrow} i \models \varphi\, \mathcal{U}_I\, \psi
\end{aligned}$$

□

Since MDL extends LDL, it can express any ω-regular language [17]. For instance, the property "the event a occurs at every even position" can be expressed as $[(\mathit{true} \cdot \mathit{true})^*]\, a$. This property cannot be expressed by any LTL or MTL formula. Similarly, the property "both b and c occur within the next two time-units and b occurs before c" cannot be expressed in MTL with its point-based semantics [12]. It can, however, be expressed in MDL with the formula $|\mathit{true}^* \cdot b \cdot \mathit{true}^* \cdot c?\rangle[0,2]$.

7 An Almost Event-Rate Independent Monitor for MDL

In our MTL monitoring algorithm, only the progress function and the computation of interval-skewed subformulas are specific to MTL. In the following subsections we will change these ingredients to obtain a monitor for MDL.

7.1 Derivatives of the Regular Expression Modalities

To build on ideas from our algorithm for MTL, we need an alternative recursive definition of the past and future modalities that refer only to the $(i-1)$st and $(i+1)$st time-point. For a fixed stream ρ, the following characterization holds.

$$i \models |r\rangle_I \text{ iff } (a = 0 \wedge \varepsilon_i(r)) \vee \Big(\Delta_i \leq b \wedge i+1 \models |\delta_i(r)\rangle_{I-\Delta_i}\Big) \tag{3}$$
$$i \models \langle r|_I \text{ iff } (a = 0 \wedge \varepsilon_i(r)) \vee \Big(i > 0 \wedge \Delta_{i-1} \leq b \wedge i-1 \models \langle \mathfrak{d}_i(r)|_{I-\Delta_{i-1}}\Big) \tag{4}$$

Here, $I = [a,b]$, $\varepsilon_i(r)$ is the Boolean denoting whether $(i,i) \in \mathcal{R}(r)$ (i.e., r matches the empty word), and $\delta_i(r)$ is the Brzozowski derivative [13] of the regular expression r (and $\mathfrak{d}_i(r)$ its symmetric counterpart). For plain regular expressions, the Brzozowski derivative $\delta_c(r)$ computes a regular expression whose language is the left quotient $\{w \mid cw \in L(r)\}$ of the input expression's language $L(r)$ by a given letter c. One may view the derivative as a

deterministic automaton whose states are labeled by regular expressions, whereby reading c in a state r takes the automaton to $\delta_c(r)$. For MDL formulas, the time-point i takes the place of the given letter c and "reading c" means querying a subformula's satisfaction at i. The inductive definitions of ε, δ, and $\mathfrak{d}$ follow. They all are implicitly parameterized by the fixed ρ.

$$\begin{aligned}
\varepsilon_i(\star) &= \bot & \delta_i(\star) &= \top? & \mathfrak{d}_i(\star) &= \top? \\
\varepsilon_i(\varphi?) &= i \models \varphi & \delta_i(\varphi?) &= \bot? & \mathfrak{d}_i(\varphi?) &= \bot? \\
\varepsilon_i(r+s) &= \varepsilon_i(r) \vee \varepsilon_i(s) & \delta_i(r+s) &= \delta_i(r) + \delta_i(s) & \mathfrak{d}_i(r+s) &= \mathfrak{d}_i(r) + \mathfrak{d}_i(s) \\
\varepsilon_i(r \cdot s) &= \varepsilon_i(r) \wedge \varepsilon_i(s) & \delta_i(r \cdot s) &= \delta_i(r) \cdot s + \varepsilon_i(r)? \cdot \delta_i(s) & \mathfrak{d}_i(r \cdot s) &= r \cdot \mathfrak{d}_i(s) + \varepsilon_i(s)? \cdot \mathfrak{d}_i(r) \\
\varepsilon_i(r^*) &= \top & \delta_i(r^*) &= \delta_i(r) \cdot r^* & \mathfrak{d}_i(r^*) &= r^* \cdot \mathfrak{d}_i(r)
\end{aligned}$$

The definition of δ is faithful to Brzozowski's original definition. Note that $\mathcal{R}(\top?) = \{(i,i) \mid i \in \mathbb{N}\}$, $\mathcal{R}(\bot?) = \{\}$, and $\varepsilon_i(r)? \cdot \delta_i(s)$ is equivalent to if $\varepsilon_i(r)$ then $\delta_i(s)$ else $\bot?$, which is more commonly used to define Brzozowski derivatives. The equations for the right derivative $\mathfrak{d}$ are symmetric for the concatenation and star cases. Thereby, $\mathfrak{d}$ matches the regular expression from right to left. It is easy to verify that the Equations 3 and 4 hold for those definitions by structural induction on the regular expression r.

How can we integrate Equations 3 and 4 into our monitor? Since the equations refer to the satisfiability of the formulas $|\delta_i(r)\rangle_{I-\Delta_i}$ and $\langle \mathfrak{d}_i(r)|_{I-\Delta_{i-1}}$, those formulas must occur in our interval-skewed subformula array. In other words, we must monitor $|\delta_i(r)\rangle_I$ simultaneously to $|r\rangle_I$ (and all their interval-skewed variants). But by the same reasoning, $|\delta_j(\delta_i(r))\rangle_I$ must be monitored, too. Hence, we must monitor all formulas that can be reached by repeatedly computing the derivative of the original subexpressions. Fortunately, Brzozowski has proved that the set of expressions reachable by repeatedly taking derivatives is finite, provided that one rewrites expressions to a normal form with respect to the associativity, commutativity, and idempotence (ACI) of the $+$ constructor. Unfortunately, the number of all such Brzozowski derivatives is exponential in the size of the initial expression r. This is not surprising since regular expressions are exponentially more concise than deterministic automata and the set of all Brzozowski derivatives represents exactly the set of states of a deterministic automaton.

With the size of the array exponential in the size of the input formula, we would still obtain an almost event-rate independent monitor, but not one that is very time efficient. We can do better by resorting to nondeterministic automata, which are as concise as regular expressions. The equivalent of the Brzozowski derivative for nondeterministic automata are Antimirov's partial derivatives of regular expressions [1]. Instead of computing only one successor expressions, a partial derivative computes a set of expressions, analogous to the transition function of a nondeterministic automaton. The partial derivative ∂ and its symmetric counterpart $\mathfrak{6}$ are defined inductively as follows.

$$\begin{aligned}
\partial_i(\star) &= \{\top?\} & \mathfrak{6}_i(\star) &= \{\top?\} \\
\partial_i(\varphi?) &= \{\} & \mathfrak{6}_i(\varphi?) &= \{\} \\
\partial_i(r+s) &= \partial_i(r) \cup \partial_i(s) & \mathfrak{6}_i(r+s) &= \mathfrak{6}_i(r) \cup \mathfrak{6}_i(s) \\
\partial_i(r \cdot s) &= \partial_i(r) \odot s \cup \varepsilon_i(r)? \odot \partial_i(s) & \mathfrak{6}_i(r \cdot s) &= r \odot \mathfrak{6}_i(s) \cup \varepsilon_i(s)? \odot \mathfrak{6}_i(r) \\
\partial_i(r^*) &= \partial_i(r) \odot r^* & \mathfrak{6}_i(r^*) &= r^* \odot \mathfrak{6}_i(r)
\end{aligned}$$

Here, $\odot$ lifts $\cdot$ to sets of expressions. Overloading notation we have $r \odot X = \{r \cdot s \mid s \in X\}$ and $X \odot r = \{s \cdot r \mid s \in X\}$.

Partial derivatives enjoy nice properties: the sum of all expressions in $\partial_i(r)$ is equivalent to $\delta_i(r)$. Moreover, the number of different expressions reachable from r by repeated application of the partial derivative is bounded by $n+1$, where n is r's size [1]. In other words, partial derivatives convert a regular expression of size n into a nondeterministic automaton of size $n+1$. The states of this automaton are labeled by the $n+1$ reachable expressions, and these are exactly the ones our monitor must keep track of to follow the following partial derivative variant of Equations 3 and 4.

$$i \models |r\rangle_I \text{ iff } (a = 0 \wedge \varepsilon_i(r)) \vee \left(\Delta_i \leq b \wedge \bigvee_{s \in \partial_i(r)} i+1 \models |s\rangle_{I-\Delta_i}\right) \tag{5}$$

$$i \models \langle r|_I \text{ iff } (a = 0 \wedge \varepsilon_i(r)) \vee \left(i > 0 \wedge \Delta_{i-1} \leq b \wedge \bigvee_{s \in \mathfrak{G}_i(r)} i-1 \models \langle s|_{I-\Delta_{i-1}}\right) \tag{6}$$

Those equations follow by structural induction on r, using the distributivity of the match operators over $+$, i.e., $i \models |r+s\rangle_I \leftrightarrow i \models |r\rangle_I \vee |s\rangle_I$.

We must know which regular expressions to keep track of before actually running the monitor (i.e., before reading ρ). We can overapproximate this set of iterated partial derivatives by replacing $\varepsilon_i(\varphi?)$ with $\top$ instead of $i \models \varphi$. Then both ε and ∂ become independent of the fixed ρ and i. Let us call this overapproximation $\hat{\partial}$ (or $\hat{\mathfrak{G}}$) for a regular expression r. The number of expressions in $\hat{\partial}(r)$ (or $\hat{\mathfrak{G}}(r)$) is bounded by the size of the original expression (+1). Our monitor keeps track of all such interval-skewed partial derivatives defined as follows:

$$\mathsf{ISF}^{\partial}(\varphi) = \mathsf{ISF}(\varphi) \cup \{\langle s|_I \mid \langle r|_I \in \mathsf{ISF}(\varphi), s \in \hat{\mathfrak{G}}(r)\} \cup \{|s\rangle_I \mid |r\rangle_I \in \mathsf{ISF}(\varphi), s \in \hat{\partial}(r)\}\}.$$

7.2 An Almost Event-Rate Independent Monitor for MDL

The recursive equations are a useful blueprint. However, we cannot use the ε and partial derivative operations directly, since they rely on the satisfiability of subformulas that are arguments of _?. But the monitor might not know at time-point i whether some subformula φ is satisfied, since φ could refer to the future. However, the monitor does know the symbolic future expression $\mathbf{curr}[\varphi]$ denoting φ's satisfiability at i. This knowledge allows us to compute the ε symbolically as a future expression:

$$\begin{array}{lll} \varepsilon(\star) = \mathsf{Now}\ \bot & \varepsilon(r+s) = \varepsilon(r) \vee_{fexp} \varepsilon(s) & \\ \varepsilon(\varphi?) = \mathbf{curr}[\varphi] & \varepsilon(r \cdot s) = \varepsilon(r) \wedge_{fexp} \varepsilon(s) & \varepsilon(r^*) = \mathsf{Now}\ \top \end{array}$$

Unlike previous definitions of ε, this definition does not depend on any fixed stream ρ.

For the symbolic version of partial derivatives, we must address the following complication. Our definition of ∂ computes a set of expressions, requiring the Boolean verdicts of certain subformulas (through ε). When working with future regular expressions, we lack the information on whether to include the partial derivatives of s when computing the partial derivatives of $r \cdot s$, since $\varepsilon(r)$ is not a Boolean value, but rather a future Boolean expression. As Equations 5 and 6 illustrate, we are not interested in regular expressions as such, but rather in expressions wrapped into some fixed past or future match operators and ultimately the satisfiability of the resulting formulas. Satisfiability queries can be represented using our machinery as future expressions. Using continuation-passing-style programming, we obtain the symbolic partial derivative ∂ (and the symmetric $\mathfrak{G}$) that computes a future expression corresponding to $\bigvee_{s \in \partial_i(r)} i+1 \models |s\rangle_{I-\Delta_i}$ (and $\bigvee_{s \in \mathfrak{G}_i(r)} i-1 \models \langle s|_{I-\Delta_{i-1}}$, respectively). The function ∂ takes two arguments: a regular expression r and a continuation function κ that, in

the base case, wraps a regular expressions in a past or future match operator and creates a variable pointing to the corresponding formula in the $(i+1)$st time-point.

$$\begin{aligned}
\partial\,(\star, \kappa) &= \kappa(\top?) & \mathfrak{6}\,(\star, \kappa) &= \kappa(\top?)\\
\partial\,(\varphi?, \kappa) &= \mathsf{Now}\,\bot & \mathfrak{6}\,(\varphi?, \kappa) &= \mathsf{Now}\,\bot\\
\partial\,(r+s, \kappa) &= \partial\,(r, \kappa) \vee_{fexp} \partial\,(s, \kappa) & \mathfrak{6}\,(r+s, \kappa) &= \mathfrak{6}\,(r, \kappa) \vee_{fexp} \mathfrak{6}\,(s, \kappa)\\
\partial\,(r \cdot s, \kappa) &= \partial\,(r, \lambda t.\, \kappa\,(t \cdot s)) \vee_{fexp} & \mathfrak{6}\,(r \cdot s, \kappa) &= \mathfrak{6}\,(s, \lambda t.\, \kappa\,(r \cdot t)) \vee_{fexp}\\
&\quad (\varepsilon(r) \wedge_{fexp} \partial\,(s, \kappa)) & &\quad (\varepsilon(s) \wedge_{fexp} \mathfrak{6}\,(r, \kappa))\\
\partial\,(r^*, \kappa) &= \partial\,(r, \lambda t.\, \kappa\,(t \cdot r^*)) & \mathfrak{6}\,(r^*, \kappa) &= \mathfrak{6}\,(r, \lambda t.\, \kappa\,(r^* \cdot t))
\end{aligned}$$

Observe how the continuation is altered in the concatenation and star cases. The standard partial derivative first calculates recursively the set $\partial_i(r)$ before concatenating s to each expression in $\partial_i(r)$. Here, we extend the continuation κ to perform the concatenation via $\lambda t.\, \kappa\,(t \cdot s)$ at the leaves of the recursion tree.

Finally, we define the progress function for MDL. As with MTL, the function takes as input a subformula φ, the time-stamp difference Δ between the current and the previous time-point, and the set of currently true atomic predicates π. Moreover, it assumes that the array **prev** contains the Boolean expressions denoting the satisfiability at the previous time-point for all interval-skewed variants of φ and that the array **curr** contains the future expression denoting the satisfiability at the current time-point for all proper subformulas of φ. It computes a future expression denoting the satisfiability of φ at the current time-point.

$$\begin{aligned}
&\mathsf{progress}\,(\varphi, \Delta, \pi) = \mathsf{case}\ \varphi\ \mathsf{of}\\
&\quad \mid p \qquad\quad \Rightarrow \mathsf{Now}\,(p \in \pi)\\
&\quad \mid \neg\psi \qquad \Rightarrow \neg_{fexp}(\mathbf{curr}[\psi])\\
&\quad \mid \psi_1 \vee \psi_2 \Rightarrow \mathbf{curr}[\psi_1] \vee_{fexp} \mathbf{curr}[\psi_2]\\
&\quad \mid |r\rangle_{[a,b]} \Rightarrow (\mathsf{Now}\,(a=0) \wedge_{fexp} \varepsilon(r)) \vee_{fexp}\\
&\qquad\qquad\quad \mathsf{Later}\,(\lambda\Delta'.\, \Delta' \leq b \wedge \mathsf{eval}_{\Delta'}(\partial\,(r, \lambda s.\, \mathsf{Now}\,(\mathsf{Var}\,(|s\rangle_{[a,b]-\Delta'})))))\\
&\quad \mid \langle r|_{[a,b]} \Rightarrow (\mathsf{Now}\,(a=0) \wedge_{fexp} \varepsilon(r)) \vee_{fexp}\\
&\qquad\qquad\quad (\mathsf{Now}\,(\Delta \leq b) \wedge_{fexp} \mathfrak{6}\,(r, \lambda s.\, \mathsf{subst}_{fexp}(\mathbf{prev}[\langle s|_{[a,b]-\Delta}]])))
\end{aligned}$$

Only the cases for the match operators are interesting. They implement Equations 5 and 6. The first disjunct is the same for both the future and past, since it covers the case when the regular expression matches the empty word. For the future match operator, the second disjunct is a Later future expression, since it does not know the time-stamp difference between the current and the next time-point. The argument to Later is the conjunction of the Boolean from the interval boundary test with the symbolic partial derivative ∂ (evaluated to a Boolean expression using the abstracted time-difference Δ'). The continuation κ wraps a given regular expression into a match formula and creates a variable denoting the satisfiability of the resulting formula at the next time-point. For the past match operator, the second disjunct is the conjunction of the interval boundary test and the right derivative $\mathfrak{6}$. The continuation function for the latter wraps a given regular expression into a match formula and retrieves the Boolean expression denoting the formula's satisfaction at the previous time-point from **prev**. The variables in this expression point to the current time-point. The function subst updates those variables to the next time-point by accessing **curr**.

Using this progress function in the algorithm shown in Section 5 results in our almost event-rate independent monitor for MDL.

7.3 Correctness

Our monitors for MTL and MDL differ only in the progress function. We establish exactly the same invariants for MDL as for MTL and obtain the correctness and complexity statements.

Theorem 4 *The monitor for a MDL formula Φ is sound: whenever it outputs the Boolean verdict $((\tau, i), b)$ 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 on bounded future formulas and its space consumption is almost event-rate independent for the* LOCAL *and* GLOBAL *modes.*

Proof The proof is analogous to MTL, including the invariants ($\mathcal{I}1$)–($\mathcal{I}6$) with $\mathsf{ISF}(\Phi)$ replaced by $\mathsf{ISF}^{\partial}(\Phi)$. We show the only new part, namely the auxiliary lemma (used to establish the preservation of ($\mathcal{I}2$) as in the MTL case) for the MDL progress function: Fix a stream $\rho = \langle(\pi_i, \tau_i)\rangle_{i\in\mathbb{N}}$ and the MDL formula Φ. Let $0 < i = \mathbf{now@off}$, $\varphi \in \mathsf{ISF}^{\partial}(\Phi)$, and $fb = \mathsf{progress}(\varphi, \Delta_{i-1}, \pi_i)$. Assume that (i) for all $\varphi \in \mathsf{ISF}(\Phi)$ we have $i-1 \models \varphi \leftrightarrow i \models_{bexp} \mathbf{prev}[\varphi]$ and (ii) for all proper subformulas ψ of φ, we have $i \models \psi \leftrightarrow i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\mathbf{curr}[\varphi])$. Then $i \models \varphi \leftrightarrow i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(fb)$.

The lemma follows by a case distinction on φ. We only show the match operator cases. They rely on several auxiliary lemmas: (ε) relating ε and ε_i via $i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\varepsilon(r)) \leftrightarrow \varepsilon_i(r)$, ($\partial$) relating ∂ and ∂_i via $i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\partial\ (r,\kappa)) \leftrightarrow \bigvee_{s\in\partial_i(r)} i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\kappa(s))$, and ($\mathfrak{G}$) relating $\mathfrak{G}$ and $\mathfrak{G}_i$ via $i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\mathfrak{G}\ (r,\kappa)) \leftrightarrow \bigvee_{s\in\mathfrak{G}_i(r)} i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\kappa(s))$ for subexpressions r of φ. These lemmas follow by induction on regular expressions using assumption (ii). We use the already familiar (from the MTL proof) property ($*$) for subst_{fexp}.

$\varphi = |r\rangle_{[a,b]}$: We have $fb = (\mathsf{Now}\ (a=0) \wedge_{fexp} \varepsilon(r)) \vee_{fexp}$
$\mathsf{Later}\ (\lambda\Delta'.\ \Delta' \le b \wedge \mathsf{eval}_{\Delta'}(\partial\ (r, \lambda s.\ \mathsf{Now}\ (\mathsf{Var}\ (|s\rangle_{[a,b]-\Delta'})))))$ and hence

$$\begin{aligned}
i \models \varphi &\overset{\text{Eq. 6}}{\leftrightarrow} (a=0 \wedge \varepsilon_i(r)) \vee \left(\Delta_i \le b \wedge \textstyle\bigvee_{s\in\partial_i(r)} i+1 \models |s\rangle_{[a,b]-\Delta_i}\right)\\
&\overset{(\varepsilon)}{\leftrightarrow} (a=0 \wedge i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\varepsilon(r))) \vee\\
&\quad \left(\Delta_i \le b \wedge \textstyle\bigvee_{s\in\partial_i(r)} i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\mathsf{Now}\ (\mathsf{Var}\ (|s\rangle_{[a,b]-\Delta_i})))\right)\\
&\overset{(\partial)}{\leftrightarrow} (a=0 \wedge i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\varepsilon(r))) \vee\\
&\quad \left(\Delta_i \le b \wedge i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\partial\ (r, \lambda s.\ \mathsf{Now}\ (\mathsf{Var}\ (|s\rangle_{[a,b]-\Delta_i}))))\right)\\
&\leftrightarrow\ i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(fb).
\end{aligned}$$

$\varphi = \langle r|_{[a,b]}$: We have $fb = (\mathsf{Now}\ (a=0) \wedge_{fexp} \varepsilon(r)) \vee_{fexp}$
$(\mathsf{Now}\ (\Delta_{i-1} \le b) \wedge_{fexp} \mathfrak{G}\ (r, \lambda s.\ \mathsf{subst}_{fexp}(\mathbf{prev}[\langle s|_{[a,b]-\Delta_{i-1}}]])))$ and hence

$$\begin{aligned}
i \models \varphi &\overset{\text{Eq. 6, } i>0}{\leftrightarrow} (a=0 \wedge \varepsilon_i(r)) \vee \left(\Delta_{i-1} \le b \wedge \textstyle\bigvee_{s\in\mathfrak{G}_i(r)} i-1 \models \langle s|_{[a,b]-\Delta_{i-1}}\right)\\
&\overset{\text{(i), }(\varepsilon)}{\leftrightarrow} (a=0 \wedge i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\varepsilon(r))) \vee\\
&\quad \left(\Delta_{i-1} \le b \wedge \textstyle\bigvee_{s\in\mathfrak{G}_i(r)} i \models_{bexp} \mathbf{prev}[\langle s|_{[a,b]-\Delta_{i-1}}]\right)\\
&\overset{(*)}{\leftrightarrow} (a=0 \wedge i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\varepsilon(r))) \vee\\
&\quad \left(\Delta_{i-1} \le b \wedge \textstyle\bigvee_{s\in\mathfrak{G}_i(r)} i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\mathsf{subst}_{fexp}(\mathbf{prev}[\langle s|_{[a,b]-\Delta_{i-1}}]))\right)\\
&\overset{(\mathfrak{G})}{\leftrightarrow} (a=0 \wedge i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\varepsilon(r))) \vee\\
&\quad \left(\Delta_{i-1} \le b \wedge i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(\mathfrak{G}\ (r, \lambda s.\ \mathsf{subst}_{fexp}(\mathbf{prev}[\langle s|_{[a,b]-\Delta_{i-1}}]))))\right)\\
&\overset{(*)}{\leftrightarrow}\ i+1 \models_{bexp} \mathsf{eval}_{\Delta_i}(fb). \qquad \square
\end{aligned}$$

8 Implementation

We have implemented the monitoring algorithms for both MTL and MDL in a tool called AERIAL [6], consisting of about 2500 lines of OCaml code. In previous work [3], we reported on a PolyML implementation of the homonymous MTL monitor. The OCaml successor incorporates optimizations, used in both logics, that substantially improve its performance.

Most of our implementation is structured into modules. The generic module Monitor implements the core logic behind almost event-rate independent monitoring, similarly to what we present in Figure 2 (right), but purely functional without global arrays. The implementation includes the three modes NAIVE, LOCAL, and GLOBAL. The Monitor module is language-independent, which supports introducing new specification languages. Building a monitor for a new language simply requires implementing a new module that specifies the language's syntactic representation and parsing and implements the language-specific progress function. We have implemented separate modules for MTL and MDL formulas. Additionally, the module Generator produces random formulas and logs given the appropriate size parameters.

A central operation in our monitor is the access to the **curr** and **prev** arrays based on a subformula's index. This raises the question of how to efficiently retrieve a subformula's index. Searching the array of subformulas is of course inefficient, although our previous PolyML implementation did just that. A more efficient solution would be to use a hash table, but preliminary experiments showed that computing formulas' hashes quickly becomes a bottleneck. Instead, AERIAL stores the indices for all subformulas directly in the formulas as annotations on the constructors. In the progress function, one can obtain the index of a formula based on the indices of its subformulas and the interval. However, the stored formula index allows us to avoid computing the indices of the subformulas recursively.

For MTL it is easy to compute the exact position of a temporal formula $\varphi\,\mathcal{U}_I\,\psi$ based on this information by using the well-order $<$ over $\mathsf{ISF}(\varphi\,\mathcal{U}_I\,\psi)$: the index of $\varphi\,\mathcal{U}_I\,\psi$ is just the index of ψ increased by b, where $I=[a,b]$. For MDL this is problematic, since the derivatives are hard to align in a predictable way. We resort to memoizing the derivative functions ∂ and $\eth$ to compute a symbolic expression not only in the verdicts at the $(i+1)$st time-point but also in the verdicts at the ith time-point. The search for indices thereby happens only once during the monitor's initialization and not during the progress function. The progress function must merely substitute the symbolic variables pointing to the ith time-point with the current values of **curr**. To further increase memory efficiency, our monitor stores only Boolean expressions for temporal subformulas. The expressions for the Boolean connectives are computed on the fly, accessing the **curr** array for temporal subformulas and, hence, are not stored.

Another crucial question is how to represent Boolean expressions stored in **curr**, **prev**, and **hist**. AERIAL offers the choice between three representations: the one reported in this paper, one based on our simple implementation of binary decision diagrams (BDDs), and one based on the BDDs implemented in the SAFA library [34]. AERIAL uses an abstract module that can be instantiated by a specific representation. This makes it easy to plug in different representations and assess their performance. Our goal in the first representation was to keep the expressions small. To achieve this, we normalize expressions with respect to the associativity, commutativity, and idempotence of $\wedge$ and $\vee$, as well as Boolean tautologies such as $\top\wedge c=c$. Boolean expressions offer a low-cost substitution operation, but they are expensive to check for equivalence. In fact, our implementation of the equivalence check translates expressions into BDDs. The BDD versions always work with the BDD representation, thereby avoiding the costly translation. In contrast, the substitution operation becomes more expensive.

Our simpler BDD representation serves as a back-up solution in cases where the richer functionality of the SAFA library imposes a performance overhead. The SAFA library uses

hash-consing [22] to maximally share the BDD structures, thereby reducing memory consumption. However, computing hashes is time-consuming. To better balance the time-space trade-off and avoid recomputation, hash-consing is often used in combination with memoization. We have not yet employed memoization in our tool. The expensive (in the BDD representation) `subst` function would be a natural candidate to memoize. However, it is technically challenging to do this efficiently as `subst` depends on the **prev** array, whose content needs to be taken into account during memoization.

9 Evaluation

In our evaluation, we consider both variants of our tool: AERIAL MTL and AERIAL MDL, implementing the MTL and MDL monitoring algorithms, respectively. We refer to AERIAL without specifying the input language for statements that apply to both monitoring algorithms. As discussed before, AERIAL allows us to specify different modes of operation that determine the contents of the **hist** array (hereafter referred to as NAIVE, LOCAL, and GLOBAL), as well as to select different representations of Boolean expressions (hereafter referred to as EXPR, BDD, and SAFA). We compare AERIAL to MONPOLY [5, 8, 9], a state-of-the-art monitor for *metric first-order temporal logic* (*MFOTL*) and MONTRE [39, 40], a state-of-the-art matcher for *timed regular expressions* (*TRE*).

Our evaluation aims to answer the following questions:

Q1: *How does* AERIAL *scale with respect to the event rate?*
Q2: *How does* AERIAL *scale with respect to the size of the monitored formula?*
Q3: *How does* AERIAL *scale with respect to the size of the formula intervals?*
Q4: *Which* AERIAL *mode (*NAIVE, LOCAL, *or* GLOBAL*) scales best?*
Q5: *Which (Boolean expression) representation (*EXPR, BDD, *or* SAFA*) scales best?*
Q6: *How does* AERIAL *perform compared to the existing state-of-the-art tools?*

We answer the above questions with three experiments attesting to the scalability of all the tools with respect to the event rate, the formula size, and the formula interval size. In our first experiment, we monitor a fixed set of formulas ($\Diamond_{[0,5]} p$, $p\,\mathcal{U}_{[0,5]}\,q$, $p\,\mathcal{U}_{[0,5]}\,(q\,\mathcal{S}_{[2,6]}\,r)$, and $p\,\mathcal{U}_{[0,5]}\,(q\,\mathcal{U}_{[2,6]}\,r)$) over streams with an increasing event rate. In our second experiment, we monitor formulas of increasing size over streams with a fixed event rate of 100 events per time unit. In our third experiment, we monitor a formula of the form $p\,\mathcal{U}_{[0,N]}\,(q\,\mathcal{S}_{[\frac{N}{2},N]}\,r)$ over streams with a fixed event rate of 100 events per time unit, while varying the value of the parameter N. All combinations of AERIAL's input languages, modes, and representations are evaluated as a family of 18 different tools.

We ran all our experiments on a machine with two sockets, each containing twelve Intel Xeon 2.20GHz CPU cores with hyperthreading. We use GNU Parallel [36] to exploit the 48 independent threads to speed up both the generation of the streams and the execution of the actual experiments. We measure the tools' total execution time and maximal memory usage using the Unix time command. Having thoroughly tested the tools' outputs separately, we discard any output during the experiments to minimize noise in our measurements.

We fix the time span of the finite prefixes of the streams used in the experiments to 100 time units, with the event rate (the number of time-points labeled with the same time-stamp) ranging from 100 to 100,000 on average ($\pm 10\%$) per stream. The streams contain three atomic propositions, $\Sigma = \{p, q, r\}$, and their distribution depends on four different generation strategies: *random*, *constant*, *custom*, and *monpoly*. The random generation strategy uses the uniform probability distribution for each event. Under the constant strategy, each stream has identical events at every time-point. Since $|\Sigma| = 3$, there are exactly eight distinct constant

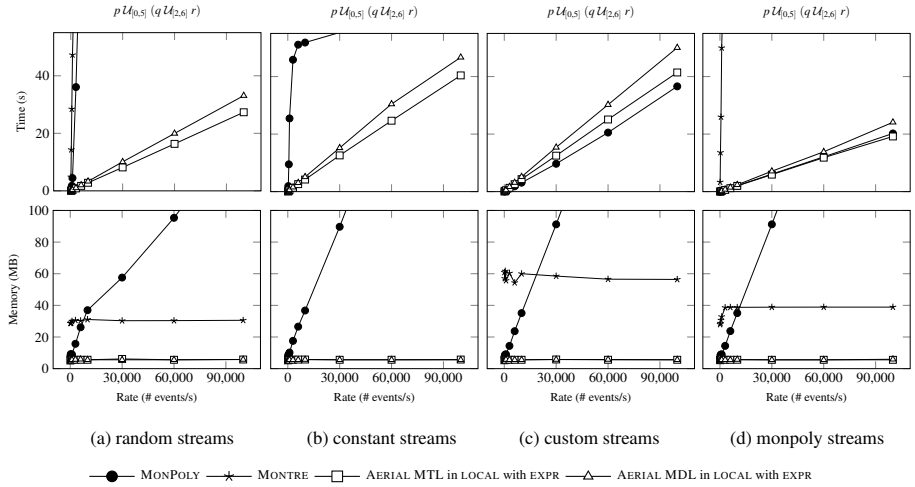
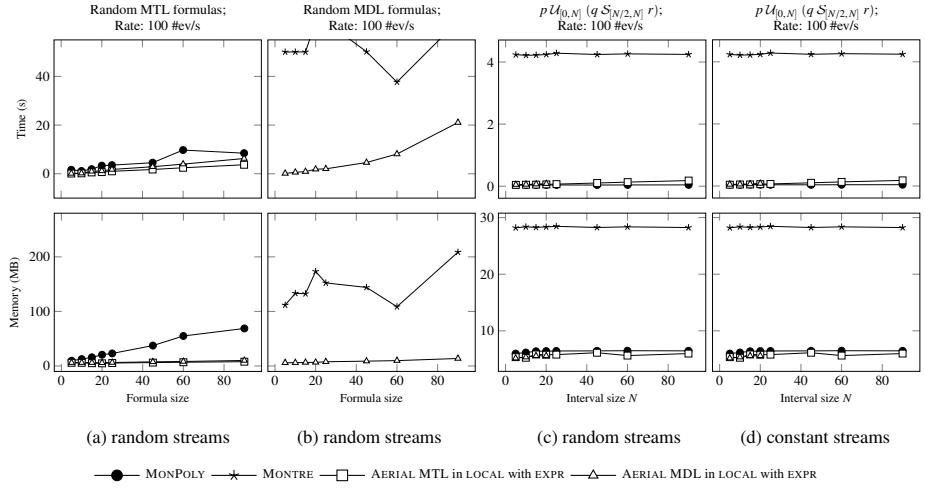


Fig. 5: Time (top) and memory (bottom) against event rate

streams, including the stream consisting of only empty time-points, and the stream consisting only of time-points in which all atomic propositions hold. Constant streams serve to test edge cases in the monitors' implementations and often trigger worst-case monitor execution time and memory usage (and the best-case as well). The custom generation strategy uses probability distributions tailored to the particular formulas. For example, for $\Diamond_{[0,5]}\,p$, the probability of p occurring is very small, which makes the tools wait longer before producing a verdict. Finally, the monpoly generation strategy uses MonPoly's own stream generator used in the evaluation [9], which we adapted to the propositional setting.

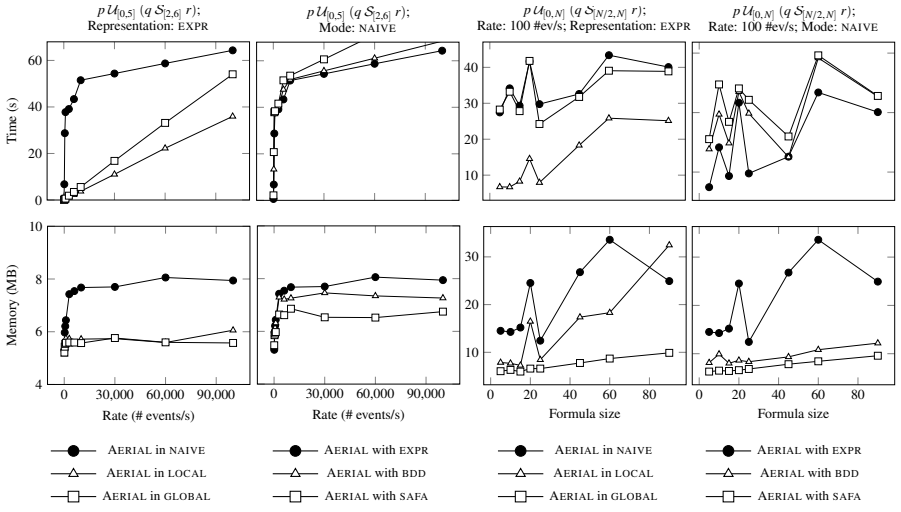
Streams generated with the custom and monpoly strategies are used only in the first experiment, since these strategies are tailored for the four formulas shown above. For each strategy and event rate, we generated eight different streams. We also converted each of these streams to the format supported by Montre. Since Montre supports only strictly monotonic time-stamps, our conversion simulated large event rates by increasing the time granularity: time-points that share the same time-stamp in the original stream are converted into a sequence of time-points with time-stamps that strictly increase by one time unit. The granularity of the time unit in the converted stream is proportional to the event rate.

Our Generator module generates random MTL and MDL formulas given a size parameter. We define the formula size as the number of subformulas. We separately check the scalability of the monitors with respect to different interval sizes in the formulas. The tools can only be compared on commonly supported logical fragments. Propositional MTL with both future and past is the common fragment supported by Aerial MDL, Aerial MTL, and MonPoly, while MDL formulas in positive normal form belong to the common fragment of Aerial MDL, and Montre. To supply the correct input to each tool, the formula generator implements a translation from MTL to fragments of MDL, MFOTL, and a translation from MDL to TRE. The translation to TRE also scales the intervals appropriately to match the different time granularity of Montre-compliant streams. In contrast to monitors that report violations, Montre outputs all parts of the stream that match a TRE pattern. Hence, to properly compare the tools, we negate the formulas provided as input to the other monitors. We generated eight arbitrary formulas for each formula size, ranging from 5 to 100.



(a) random streams (b) random streams (c) random streams (d) constant streams

Fig. 6: Time (top) and memory (bottom) usage against formula (left) and interval size (right)



(a) constant streams (b) constant streams (c) constant streams (d) constant streams

Fig. 7: Time (top) and memory (bottom) usage of Aerial MDL's modes and representations

We set a timeout for each monitoring run to be 100 seconds, coinciding with the streams' time span. Moreover, we employ the following disqualification rule: If a tool times out for all the runs with some event rate (or some formula size) then it will not be invoked with streams with larger event rates (or with larger formula sizes). When computing the mean values, a timeout counts as 100 seconds (although the actual run may take longer) and skews the curves to converge to the 100 second margin. Therefore, in our plots we only show values below 50 seconds. The memory used before a timeout contributes to the mean memory usage.

Figure 5 shows the results of the first part of the evaluation classified according to the stream generation strategy. We show the plots for formula $p\,\mathcal{U}_{[0,5]}\ (q\,\mathcal{U}_{[2,6]}\ r)$, which had the least favorable outcome for our tool. Each data point in the plots shows the average calculated

over eight different streams with a fixed event rate. To answer Q1, scalability with respect to the event rate, observe that the space consumption of both versions of AERIAL is constant. As expected, the increasing memory consumption of other tools significantly increases the overall processing time. MONTRE was almost immediately disqualified in the case of constant streams. Its plot is not visible in the case of custom streams due to many timeouts.

To answer Q2, scalability with respect to formula size, note that, even for the largest formula, AERIAL requires only 12MB of space compared to 100MB used by MONPOLY, (Figure 6a) and 250MB used by MONTRE (Figure 6a). These experiments were performed on random traces and random formulas and each data point is an average value over eight random traces and eight random formulas with the same size. During the experiment shown in Figure 6b, MONTRE timed out 3,816 times which is over 82% of all its invocations. Figures 6c and 6d answer Q3 and show that all the tools are mostly unaffected by the size of the time interval N in the formula $p\,\mathcal{U}_{[0,N]}\,(q\,\mathcal{S}_{[N/2,N]}\,r)$. Here each point shows the average value over eight traces with the event rate of 100 events per time unit.

Figure 7 compares performance of different versions of AERIAL, answering Q4 and Q5. Figures 7a and 7c show the scalability of the three AERIAL modes with respect to event rate and formula size respectively. As expected, the NAIVE mode scales poorly with the increasing event rate. In terms of the time efficiency, LOCAL mode is only marginally better than the GLOBAL mode. In 86% of the experiments, there was no conclusive winner between the two modes, however, we report on the most significant difference exhibited while monitoring constant streams. GLOBAL mode uses less memory, since it always removes equivalent expressions from the **hist** array. Figures 7b and 7d show the scalability of AERIAL using the three different representations for Boolean expressions with respect to event rate and formula size respectively. Note that the NAIVE mode shown in Figure 7b often times out for event rates higher then 1000, which skews the plots. Direct representation (EXPR) is the fastest overall, while the SAFA BDDs provided the best memory efficiency. This perfectly corroborates our intuition on time and memory tradeoffs when hash-consing the BDD structures.

Finally, regarding Q6, in all experiments AERIAL MDL performs only marginally worse than AERIAL MTL, while supporting a more expressive logic. Overall, AERIAL outperformed the other state-of-the-art monitoring tools both in terms of time and memory efficiency.

10 Conclusion

We have introduced the notion of event-rate independence for measuring the space complexity of online monitoring algorithms. This notion is fundamental for monitors that process event streams of varying velocity. We then presented two novel monitoring algorithms. The first is a monitor for metric temporal logic (MTL) that is almost event-rate independent. Afterwards, we introduced metric dynamic logic (MDL) and we extended our almost event-rate independent monitoring algorithm for MTL to support MDL. Our evaluation shows that our implementation of both monitors outperform other state-of-the-art monitoring tools.

As future work, we would like to extend these ideas to the first-order setting where events may carry data and formulas may quantify over the data's domain.

Acknowledgment. This research is supported by the Swiss National Science Foundation grant Big Data Monitoring (167162) and by the US Air Force grant Monitoring at Any Cost (FA9550-17-1-0306). The authors are listed alphabetically. Felix Klaedtke showed us an example of a property not expressible in MTL. Joshua Schneider participated in discussions about simplifying MDL's syntax. Jasmin Blanchette, Domenico Bianculli, and anonymous TACAS and RV reviewers us helped to improve the presentation of this work.

References

1. Antimirov, V.: Partial derivatives of regular expressions and finite automaton constructions. Theor. Comput. Sci. **155**(2), 291–319 (1996)
2. Asarin, E., Caspi, P., Maler, O.: Timed regular expressions. J. ACM **49**(2), 172–206 (2002)
3. Basin, D., Bhatt, B., Traytel, D.: Almost event-rate indepedent monitoring of metric temporal logic. In: A. Legay, T. Margaria (eds.) TACAS 2017, *LNCS*, vol. 10206, pp. 94–112. Springer (2017)
4. Basin, D., Klaedtke, F., Zălinescu, E.: Algorithms for monitoring real-time properties. Acta Informatica (2017)
5. Basin, D., Klaedtke, F., Zalinescu, E.: The MonPoly monitoring tool. In: G. Reger, K. Havelund (eds.) RV-CuBES 2017, *Kalpa Publications in Computing*, vol. 3, pp. 19–28. EasyChair (2017)
6. Basin, D., Krstic, S., Traytel, D.: AERIAL: Almost event-rate independent algorithms for monitoring metric regular properties. In: G. Reger, K. Havelund (eds.) RV-CuBES 2017., *Kalpa Publications in Computing*, vol. 3, pp. 29–36. EasyChair (2017)
7. Basin, D., Krstić, S., Traytel, D.: Almost event-rate indepedent monitoring of metric dynamic logic. In: S. Lahiri, G. Reger (eds.) RV 2017, *LNCS*, vol. 10548, pp. 85–102. Springer (2017)
8. Basin, D.A., Klaedtke, F., Müller, S., Pfitzmann, B.: Runtime monitoring of metric first-order temporal properties. In: R. Hariharan, M. Mukund, V. Vinay (eds.) FSTTCS 2008, *LIPIcs*, vol. 2, pp. 49–60. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2008)
9. Basin, D.A., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015)
10. Basin, D.A., Klaedtke, F., Zălinescu, E.: Algorithms for monitoring real-time properties. In: S. Khurshid, K. Sen (eds.) RV 2011, *LNCS*, vol. 7186, pp. 260–275. Springer (2012)
11. Bauer, A., Küster, J., Vegliach, G.: From propositional to first-order monitoring. In: A. Legay, S. Bensalem (eds.) RV 2013, *LNCS*, vol. 8174, pp. 59–75. Springer (2013)
12. Bouyer, P., Chevalier, F., Markey, N.: On the expressiveness of TPTL and MTL. Inf. Comput. **208**(2), 97–116 (2010)
13. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM **11**(4), 481–494 (1964)
14. Dax, C., Klaedtke, F., Lange, M.: On regular temporal logics with past. Acta Inf. **47**(4), 251–277 (2010)
15. De Giacomo, G., De Masellis, R., Grasso, M., Maggi, F.M., Montali, M.: LTLf and LDLf monitoring: A technical report. CoRR **abs/1405.0054** (2014)
16. De Giacomo, G., De Masellis, R., Grasso, M., Maggi, F.M., Montali, M.: Monitoring business metaconstraints based on LTL and LDL for finite traces. In: S.W. Sadiq, P. Soffer, H. Völzer (eds.) BPM 2014, *LNCS*, vol. 8659, pp. 1–17. Springer (2014)
17. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: F. Rossi (ed.) IJCAI-13, pp. 854–860. AAAI Press (2013)
18. Du, X., Liu, Y., Tiu, A.: Trace-length independent runtime monitoring of quantitative policies in LTL. In: N. Bjørner, F. de Boer (eds.) FM 2015, *LNCS*, vol. 9109, pp. 231–247. Springer (2015)
19. Faymonville, P., Finkbeiner, B., Schwenger, M., Torfah, H.: Real-time stream-based monitoring. CoRR **abs/1711.03829** (2017)
20. Faymonville, P., Zimmermann, M.: Parametric linear dynamic logic. In: A. Peron, C. Piazza (eds.) Proceedings 5th GandALF 2014, *EPTCS*, vol. 161, pp. 60–73 (2014)
21. Faymonville, P., Zimmermann, M.: Parametric linear dynamic logic. Inf. Comput. **253**, 237–256 (2017)
22. Filliâtre, J., Conchon, S.: Type-safe modular hash-consing. In: A. Kennedy, F. Pottier (eds.) ACM Workshop on ML, pp. 12–19. ACM (2006)
23. Fischer, M.J., Ladner, R.E.: Propositional dynamic logic of regular programs. J. Comput. Syst. Sci. **18**(2), 194–211 (1979)
24. Furia, C.A., Spoletini, P.: Bounded variability of metric temporal logic. In: A. Cesta, C. Combi, F. Laroussinie (eds.) TIME 2014, pp. 155–163. IEEE Computer Society (2014)
25. Gunadi, H., Tiu, A.: Efficient runtime monitoring with metric temporal logic: A case study in the Android operating system. In: C.B. Jones, P. Pihlajasaari, J. Sun (eds.) FM 2014, *LNCS*, vol. 8442, pp. 296–311. Springer (2014)
26. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: J. Katoen, P. Stevens (eds.) TACAS 2002, *LNCS*, vol. 2280, pp. 342–356. Springer (2002)
27. Henriksen, J.G., Thiagarajan, P.: Dynamic linear time temporal logic. Ann. Pure Appl. Logic **96**(1), 187–207 (1999)
28. Ho, H., Ouaknine, J., Worrell, J.: Online monitoring of metric temporal logic. In: B. Bonakdarpour, S.A. Smolka (eds.) RV 2014, *LNCS*, vol. 8734, pp. 178–192. Springer (2014)
29. Kapoutsis, C.A.: Removing bidirectionality from nondeterministic finite automata. In: J. Jedrzejowicz, A. Szepietowski (eds.) MFCS 2005, *LNCS*, vol. 3618, pp. 544–555. Springer (2005)

30. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. **2**(4), 255–299 (1990)
31. Leucker, M., Sánchez, C.: Regular linear temporal logic. In: C.B. Jones, Z. Liu, J. Woodcock (eds.) ICTAC 2007, *LNCS*, vol. 4711, pp. 291–305. Springer (2007)
32. Maler, O., Nickovic, D., Pnueli, A.: Real time temporal logic: Past, present, future. In: P. Pettersson, W. Yi (eds.) FORMATS 2005, *LNCS*, vol. 3829, pp. 2–16. Springer (2005)
33. McAfee, A., Brynjolfsson, E.: Big data: The management revolution. Harvard Business Review **90**(10), 61–67 (2012)
34. Pous, D.: Symbolic algorithms for language equivalence and Kleene algebra with tests. In: D. Walker (ed.) POPL 2015, pp. 357–368. ACM (2015)
35. Sánchez, C., Leucker, M.: Regular linear temporal logic with past. In: G. Barthe, M.V. Hermenegildo (eds.) VMCAI 2010, *LNCS*, vol. 5944, pp. 295–311. Springer (2010)
36. Tange, O.: GNU Parallel - the command-line power tool. ;login: The USENIX Magazine **36**(1), 42–47 (2011). DOI http://dx.doi.org/10.5281/zenodo.16303. URL `http://www.gnu.org/s/parallel`
37. Thati, P., Roşu, G.: Monitoring algorithms for metric temporal logic specifications. Electr. Notes Theor. Comput. Sci. **113**, 145–162 (2005)
38. Ulus, D.: Montre: A tool for monitoring timed regular expressions. arXiv preprint arXiv:1605.05963 (2016)
39. Ulus, D., Ferrère, T., Asarin, E., Maler, O.: Timed pattern matching. In: A. Legay, M. Bozga (eds.) FORMATS 2014, *LNCS*, vol. 8711, pp. 222–236. Springer (2014)
40. Ulus, D., Ferrère, T., Asarin, E., Maler, O.: Online timed pattern matching using derivatives. In: M. Chechik, J.F. Raskin (eds.) TACAS 2016, *LNCS*, vol. 9636, pp. 736–751. Springer (2016)
41. Vardi, M.Y.: From Church and Prior to PSL. In: O. Grumberg, H. Veith (eds.) 25 Years of Model Checking - History, Achievements, Perspectives, *LNCS*, vol. 5000, pp. 150–171. Springer (2008)
42. Wolper, P.: Temporal logic can be more expressive. Inform. Control **56**(1/2), 72–99 (1983)