Multi-Head Monitoring of Metric Dynamic Logic
(Extended Report)

Martin Raszyk, David Basin, and Dmitriy Traytel

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

Abstract. We develop a monitoring algorithm for metric dynamic logic, an extension of metric temporal logic with regular expressions. The monitor computes whether a given formula is satisfied at every position in an input trace of time-stamped events. Our monitor follows the multi-head paradigm: it reads the input simultaneously at multiple positions and moves its reading heads asynchronously. This mode of operation results in unprecedented space complexity guarantees for metric dynamic logic: The monitor's memory consumption neither depends on the event-rate, i.e., the number of events within a fixed time-unit, nor on the numeric constants occurring in the quantitative temporal constraints in the given formula. We formally prove our algorithm correct in the Isabelle proof assistant, integrate it in the Hydra monitoring tool, and empirically demonstrate its strong performance.

1 Introduction

In runtime verification, monitoring is the task of detecting whether a system execution trace adheres to a given specification. One typically distinguishes online monitors that observe the trace event-wise as the system's execution proceeds from offline monitors that read the recorded trace from a log file, possibly after the system has finished its execution.

We have recently proposed third mode of operation for monitors: multi-head monitoring [20, 22]. Conceptually, a multi-head monitor has multiple pointers, called reading heads, into a single log file. The reading heads move over the file, independently of each other. In contrast to an offline monitor's random access to the log, a multi-head monitor's heads are restricted to move only in one direction, from left to right. Thus, an online monitor can be seen as the special case of a multi-head monitor that uses a single head.

In our previous work [20], we have demonstrated the benefits of multi-head monitoring for metric temporal logic (MTL) [17]. MTL is a widely used propositional specification language capable of expressing qualitative (e.g., happens before) and quantitative (e.g., within the last hour) temporal relationships. Our multi-head MTL monitor supports arbitrarily nested past and bounded future operators and produces a stream of Boolean verdicts denoting the formula's satisfaction (or violation) at each position in the trace. The monitor uses as many reading heads as there are leaves in the formula's syntax tree. Its worst-case memory consumption is linear in the formula's *temporal size*, which is the sum of the formula's *size* (number of operators) and all *metric constants* occurring in the formula (the boundaries of intervals expressing quantitative temporal relationships). However, the monitor is *event-rate independent* [1], i.e., its space complexity does not depend on the trace length, the event rate, or other trace characteristics (assuming registers to store numbers as the underlying model of computation). The strong theoretical guarantees for our multi-head MTL monitor translate into practice: the monitor's implementation significantly outperforms its competitors with respect to both memory usage and the average time spent processing an event.

In this paper, we continue our investigation of the multi-head paradigm. We improve over our MTL monitor along three axis: (1) we consider a more expressive specification language than MTL, (2) we generalize the time domain to support both dense and discrete time, and (3) we achieve a space complexity that no longer depends on the metric constants occurring in the formula (again assuming the register model). As our specification language, we use metric dynamic logic (MDL) [1] (Section 2), an extension of MTL with regular expressions. The use of regular expressions instead of MTL's temporal operators increases the logic's expressiveness, which has prompted de Giacomo and Vardi to advocate linear dynamic logic (MDL's non-metric variant) over linear temporal logic [10].

Our main contribution is a space-efficient multi-head MDL monitor. On a high-level (Section 3), it resembles our multi-head MTL monitor [20]. In both logics, the main challenge for space-efficiency stems from the presence of both past and future operators, which may require the monitor to buffer the verdicts from the recursive subformula evaluation until a verdict for the overall formula can be produced. For MTL, the key insight is that a multi-head monitor can compress the information needed to evaluate MTL's temporal operators due to the simple fixed patterns of the direct subformulas' verdicts that the MTL semantics enforces. In contrast, MDL's regular expressions yield patterns that are neither simple nor fixed. We develop a data structure, called a *window*, that supports the space-efficient compression for this general case (Section 4). Consequently, our monitor is the first event-rate independent algorithm for MDL that outputs a stream of Boolean verdicts. Moreover, our new data structure's time and space complexity is independent of the formula's metric constants, a property we call *interval-obliviousness*, which the MTL monitor does not offer. Interval-obliviousness is relevant: large constants like 259 200 (three days expressed in seconds) often occur in realistic specifications [2,3].

The improvements over the multi-head MTL monitor come at a price: our MDL monitor's space consumption depends exponentially on the formula size. This follows alone from the fact that we will construct deterministic automata (on the fly) from the regular expressions occurring in the formula. Similarly, the number of required reading heads may be exponential in the formula size. In practice, however, specifications are small, while the traces are huge. It usually poses no problem for monitors to be exponential in the formula size, whereas a linear dependence on the trace or on the large numeric constants occurring in the formula is prohibitive. Our empirical evaluation of our multi-head MDL monitor confirms this "monitoring folk wisdom" (Section 5).

We used the Isabelle proof assistant to verify our monitor's functional correctness [21]. We proved its time and space complexity bounds on paper (Sect. 4.5).

Related Work Event-rate independence is impossible to achieve for single-head monitors that support past and future temporal operators and output Boolean verdicts for every position in the trace (as we argue in Section 3.3). The multi-head paradigm overcomes this limitation for MTL [20]. Recently, we have used the multi-head model of computation to eliminate non-determinism from functional finite-state transducers [22]. This theoretical result provides a stepping stone towards our multi-head MDL monitor. Our core data structure resembles the multi-head transducer for the *all-suffix regular matching* problem studied in that work. However, significant extensions were necessary to handle quantitative temporal constraints, past operators, and the arbitrary nesting of formulas and regular expressions; these are all aspects not present in the transducer setting.

An alternative approach to achieving event-rate independence is to relax the requirement to output Boolean verdicts. Instead, an out-of-order mixture of Boolean and equivalence verdicts can be used to denote that the verdict is presently unknown, but will be equivalent to some other (also presently unknown) verdict [1]. This relaxation resulted in Aerial [7], the first event-rate independent MDL monitor. Our algorithm produces much more intelligible output, while also being event-rate independent. Moreover, Aerial's space and per-event-time complexity depend linearly on the sum of the formula's metric constants, whereas our monitor is interval-oblivious. This weakness of Aerial was also observed and improved upon empirically in the Reelay monitor for past-only MTL [25]. Reelay's space complexity, however, is still linear in the sum of the formula's constants.

Stream runtime verification (SRV) [24], pioneered by LOLA [9], generalizes logic-based specifications to recursive programs using stream expressions. Some specifications expressed in these languages can be efficiently monitored in constant space, but this fragment is rather restricted: specifications may refer to a bounded number of future events and the bound must be fixed statically. In contrast, MTL's and MDL's metric constraints, even if bounded, may require the monitor to wait for an unbounded number of future events before being able to output a verdict for an earlier position. (Metric constraints bound time, which is different from counting events.) Metric extensions of SRV languages were recently proposed [8, 11, 12]. They inherit the restricted efficiently monitorable fragment from non-metric SRV languages. A similar restriction applies to quantified regular expressions [18], which can be evaluated in constant space, but support neither metric constraints nor dependencies on future events.

Beyond propositional specification languages, first-order monitors [4, 13, 15], implemented in tools like MonPoly [6] and DejaVu [14], also produce streams of verdicts. Event-rate independence is however out of reach for these algorithms [4].

2 Metric Dynamic Logic

We recapitulate metric dynamic logic (MDL) [1]. While previous works on MDL focused on natural numbered time-stamps, we consider an abstract time domain $\mathbb{T}$. We assume that $\mathbb{T}$ forms an additive commutative monoid $(\mathbb{T}, +, 0)$, a partial order $(\mathbb{T}, <)$, and a join-semilattice $(\mathbb{T}, \sqcup)$. The partial order must be consistent with $\sqcup$ and $+$, i.e., $a \leq a \sqcup b$, $b \leq a \sqcup b$, $a \leq c \wedge b \leq c \implies a \sqcup b \leq c$, and $b < c \implies a + b < a + c$, for all $a, b, c \in \mathbb{T}$. Moreover, we assume the existence of an order-preserving embedding ι of natural numbers into $\mathbb{T}$ satisfying $\forall \tau \in \mathbb{T}.\ \exists n \in \mathbb{N}.\ \tau < \iota(n)$. For example, these assumptions are satisfied by both the discrete natural numbers $\mathbb{T} = \mathbb{N}$ and the dense real numbers $\mathbb{T} = \mathbb{R}$.

Further, let $\mathbb{I}$ be the set of non-empty intervals over $\mathbb{T}$. We write $\mathbb{I}$'s elements as $[l, r]$, where $l \in \mathbb{T}$, $r \in \mathbb{T} \cup \{\infty\}$, $l \leq r$, and $[l, r] = \{x \in \mathbb{T} \mid l \leq x \leq r\}$. We also define the operation of *shifting* an interval $[l, r] \in \mathbb{I}$ by a time-stamp $\tau \in \mathbb{T}$ as $\tau + [l, r] = [\tau + l, \tau + r]$. An event stream $\rho = \langle(\pi_i, \tau_i)\rangle_{i \in \mathbb{N}}$ is an infinite sequence of sets of atomic propositions $\pi_i \subseteq \Sigma$ along with their time-stamps $\tau_i \in \mathbb{T}$, which is monotone ($\forall i.\ \tau_i \leq \tau_{i+1}$) and progressing ($\forall\tau.\ \exists i.\ \tau < \tau_i$). The event stream's indices $i \in \mathbb{N}$ are called time-points. Consecutive time-points may carry the same time-stamp, and there might be time-stamps that no time-point carries. MDL's syntax is defined as follows, where $p \in \Sigma$ and $I \in \mathbb{I}$.

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

Aside from Boolean operators, MDL contains the regular expression modalities. The future match operator $|r\rangle_I$ expresses that there exists some future time-point j whose time-stamp is in the interval $\tau + I$, where τ is the current time-point's time-stamp, and the regular expression r matches the portion of the event stream from the current point up to j. The past match operator $\langle r|_I$ expresses the dual property about a past time-point. Regular expressions themselves may nest arbitrary MDL formulas via the _? operator. We call the subformulas φ occurring as $\varphi?$ in a regular expression r the *direct tests* of r, thereby excluding any further _? operators that occur in φ itself. 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 alternation, concatenation, and (Kleene) star operators.

We define the point-based semantics [5] of formulas and regular expressions by mutual recursion. A formula is evaluated over a fixed event stream $\rho = \langle(\pi_i, \tau_i)\rangle_{i\in\mathbb{N}}$ at a time-point $i \in \mathbb{N}$. We write $i \models \varphi$ if φ is true at i, whereby we omit the explicit reference to ρ. The regular expression r's semantics for a fixed ρ is a relation $\mathcal{R}(r) \subseteq \mathbb{N}\times\mathbb{N}$, where $(i, j) \in \mathcal{R}(r)$ are the starting and ending time-points of a match. Overloading notation, $\cdot$ and $\_^*$ denote relation composition and the reflexive transitive closure.

$$\begin{array}{lll} i \models p & \text{iff } p \in \pi_i & \mathcal{R}(\star) = \{(i, i+1) \mid i \in \mathbb{N}\} \\ i \models \neg\varphi & \text{iff } i \not\models \varphi & \mathcal{R}(\varphi?) = \{(i, i) \mid i \models \varphi\} \\ i \models \varphi \vee \psi & \text{iff } i \models \varphi \vee i \models \psi & \mathcal{R}(r+s) = \mathcal{R}(r) \cup \mathcal{R}(s) \\ i \models |r\rangle_I & \text{iff } \exists j \geq i.\ \tau_j \in \tau_i + I \wedge (i, j) \in \mathcal{R}(r) & \mathcal{R}(r \cdot s) = \mathcal{R}(r) \cdot \mathcal{R}(s) \\ i \models \langle r|_I & \text{iff } \exists j \leq i.\ \tau_i \in \tau_j + I \wedge (j, i) \in \mathcal{R}(r) & \mathcal{R}(r^*) = \mathcal{R}(r)^* \end{array}$$

We assume that intervals $[l, r]$ of future match operators are bounded, i.e., $r \neq \infty$, and employ the usual syntactic sugar for additional constructs: $\mathit{true} = p \vee \neg p$, $\mathit{false} = \neg\mathit{true}$, and $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$. Given formulas φ and ψ, we define the MTL operators next $\bigcirc_I\, \varphi$ as $|\star \cdot \varphi?\rangle_I$, previous $\bullet_I\, \varphi$ as $\langle \varphi? \cdot \star|_I$, until $\varphi\, \mathsf{U}_I\, \psi$ as $|(\varphi? \cdot \star)^* \cdot \psi?\rangle_I$, and since $\varphi\, \mathsf{S}_I\, \psi$ as $\langle \psi? \cdot (\star \cdot \varphi?)^*|_I$. These abbreviations faithfully implement MTL's point-based semantics.

Example 1. Many systems for user authentication follow a policy like: "A user should not be able to authenticate after entering a wrong password three times within the last hour without successfully authenticating in between." For a fixed user, we write ✗ for the event "User entered a wrong password" and ✓ for "User has successfully authenticated." Additionally, we abbreviate $\varphi? \cdot \star$ by φ. (This abbreviation is only used when φ appears in a regular expression position, e.g., as an argument of $\cdot$ or $\_^*$). Then the MDL formula $✓ \wedge \langle\big(✗ \cdot (\neg✓)^* \cdot ✗ \cdot (\neg✓)^* \cdot ✗ \cdot (\neg✓)^*\big)|_{[0,3600]}$ captures this policy's violations: it is satisfied at time-points at which the fixed user successfully authenticated after entering wrong credentials three times in the last 3 600 seconds, without intermediate successful authentications. We can express this property in MTL by nesting six temporal operators, namely one since and one previous operator for each of the ✗ subformulas. Yet, it is unclear which intervals to use as arguments to these operators beyond the fact that their upper bounds should sum up to 3 600. For $\mathbb{T} = \mathbb{N}$, a rather impractical solution exploits that there are finitely many ways to split the interval $[0, 3600]$ and constructs the disjunction of all possible splits, which yields $\binom{3605}{5} = 5\,059\,876\,272\,308\,221$ disjuncts in this case. For $\mathbb{T} = \mathbb{R}$, the previous solution no longer works and we conjecture that no equivalent MTL formula exists. MDL remediates these difficulties regardless of the time domain.

3 High-Level Overview

Our multi-head MDL monitor follows the monitored formula's recursive structure. We describe below the main ideas for propositions, Boolean, and temporal match operators.

3.1 Propositions and Boolean Operators

For an atomic proposition, a one-head monitor scans the trace and returns the corresponding Boolean verdicts. We view non-atomic formulas as being evaluated on streams of Boolean verdicts produced by submonitors for their subformulas. For $\varphi \vee \psi$, we evaluate $b_\varphi \vee b_\psi$ over the atomic propositions b_φ and b_ψ, which denote the satisfaction of φ and ψ at each time-point. The monitor for $\varphi \vee \psi$ uses a single head to combine its inputs b_φ and b_ψ at each time-point based on the semantics of $\vee$. Negation is evaluated similarly.

3.2 Temporal Match Operators

For a formula φ of the form $|r\rangle_I$ or $\langle r|_I$, we first convert r into an automaton over the alphabet $\mathbb{B}^k$, where k is the number of r's direct tests. For each time-point, the automaton's input symbol is constructed from k Boolean verdicts for r's direct tests at this time-point.

Key to our work is a data structure, called a *window*, that maintains a summary of the automaton runs on a finite subword of the automaton's input stream. The subword starts at a position i and ends at j. For a future match formula $\varphi = |r\rangle_I$, the position i is the time-point at which we need to produce φ's next Boolean verdict and j is a suitable lookahead time-point, determined by φ's interval I, which makes it possible to evaluate φ. Note that i and j can be arbitrarily far apart, but the window's size does not depend on this distance.

For a past match formula $\varphi = \langle r|_{[a,b]}$, the verdicts are computed at the window's end j. The window's start i is the earliest time-point with $\tau_j \notin \tau_i + [a, \infty]$ or it equals j if $a = 0$. The data structure uses two reading heads, a *start head* at i and an *end head* at j, to support operations that advance the window's start and end. Advancing the window's start requires a third auxiliary reading head that is obtained by cloning the start head. As with all reading heads, this additional head may move asynchronously after cloning.

Finally, the multi-head monitor M for the temporal match formula φ maintains the window data structure and uses it to compute the Boolean verdicts for φ. To assemble the next input symbol for the automaton, M runs k submonitors for r's direct tests. In particular, a reading head of the window data structure corresponds to the states of the k submonitors and thus cloning the reading head means cloning these submonitors.

3.3 Relation to our Multi-Head Monitor for MTL

Our multi-head MTL monitor [20] coincides with our MDL monitor except for the temporal operator cases. For MTL, we use a different data structure that only requires a single reading head per temporal operator. This is possible due to the special form of the regular expressions corresponding to MTL's operators. Although simpler, the MTL data structure is not interval-oblivious. Moreover, its time-stamps are fixed to the natural numbers.

In more detail, for since and until, the MTL monitor's state contains all time-stamp differences of relevant (for the interval) past or future matches. These time-stamp differences are stored compactly to avoid a linear dependence on the trace length. Yet, the number of stored time-stamp differences depends on the interval bounds.

For the until operator $\varphi \mathsf{U}_I \psi$, producing a Boolean verdict at a time-point is delayed as long as all time-points satisfy φ and no time-point within the interval satisfies ψ. Nevertheless, all delayed time-points with the same time-stamp are guaranteed to be resolved to the same Boolean verdict. Hence, our MTL monitor stores only the number of delayed time-points for each time-stamp relevant for the interval. For MDL, it no longer holds that all delayed time-points with the same time-stamp must resolve to the same Boolean verdict. To see this, consider the formula $|\varphi? \cdot (\star)^* \cdot \psi?\rangle_{[0,0]}$, which holds at time-point i iff φ holds at i and ψ holds at some time-point $j \geq i$ with $\tau_i = \tau_j$. Producing a Boolean verdict at a time-point i for this formula must be delayed as long as no time-point j with the same time-stamp $\tau_j = \tau_i$ satisfies ψ. But if there exists such a time-point j, then all delayed time-points k, for $i \leq k \leq j$, are resolved to *true* iff φ is satisfied at k. Hence, the information to compute the Boolean verdicts for the delayed time-points cannot be compressed sublinearly with respect to the event rate. Our remedy is to use multiple reading heads, i.e., to run two monitors for φ and ψ, which process the time-points asynchronously.

4 Evaluating Temporal Match Operators

We now formally define the multi-head monitors for the past and future temporal match formulas $\langle r|_I$ and $|r\rangle_I$. First, we focus on a fixed regular expression r independently of both the interval I and whether r is used in a past or future match.

Let k be the number of direct tests of r and let ψ_j, for all $1 \leq j \leq k$, be the j-th direct test of r (according to some formula ordering). The i-th input symbol $b^i \in \mathbb{B}^k$ of the automaton, defined formally in Section 4.1, reflects the formula ψ_j's satisfaction at time-point i, i.e., b^i_j iff $i \models \psi_j$. To compute the input symbol b^i, a multi-head submonitor is run for each formula ψ_j, i.e., k synchronous multi-head monitors are run to compute b^i.

Our window data structure, defined formally in Section 4.2, reads the input symbols with multiple one-way reading heads. It has two heads positioned at the window's start and end. Advancing a head to the next time-point means advancing the corresponding k submonitors to the next time-point and assembling the next input symbol from their k Boolean verdicts. To update the window's state, a monitor may clone and advance the head at the window's start to read subsequent input symbols. Cloning does not affect the original reading head, i.e., there are always two heads at the window's start and end.

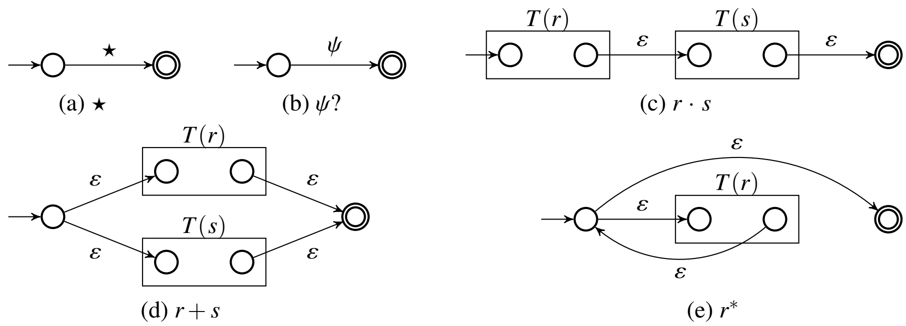


Fig. 1: Recursive conversion of an MDL regular expression into $\mathcal{A}_N$'s transition graph

4.1 Translating Regular Expressions

We first convert MDL's regular expressions into nondeterministic automata with ε-transitions over an alphabet of vectors $b^i \in \mathbb{B}^k$. A slight peculiarity, due to MDL's semantics, requires our automata to consider the current input symbol even in ε-transitions. More precisely, a regular expression $\psi?$ always matches at most a *single* time-point, i.e., according to its semantics, only pairs of the form (i, i) are included in $\mathcal{R}(\psi?)$. In particular, even the regular expression $\psi? \cdot \varphi?$ matches at most a single time-point i, specifically $(i, i) \in \psi? \cdot \varphi?$ iff $i \models \psi$ and $i \models \varphi$. Matching such an expression therefore does not consume an input symbol. In contrast, matching the regular expression $\star$ is independent of the current input symbol b^i, but always consumes an input symbol.

A textbook ε-NFA's transitions are labeled by an input symbol or ε. In contrast, we distinguish three types of edges in the transition graph of our ε-NFA:

- *conditional ε-transition labeled by ψ_j*: observes the current input symbol b^i and can be taken if $b^i_j = \mathit{true}$; does not consume an input symbol;
- *unconditional ε-transition*: can always be taken; does not consume an input symbol;
- *$\star$-transition*: can always be taken; consumes the current input symbol.

To construct the transition graph, we use Thompson's standard construction mildly adapted to MDL regular expressions and the three types of edges in the transition graph. To this end, Figure 1 defines a recursive function T on MDL regular expressions that computes the transition graph of a regular expression together with the initial and accepting state.

Because our window data structure described in the next section requires a deterministic automaton, we further determinize the obtained ε-NFA $\mathcal{A}_N$ using the subset construction. A difficulty arises from the conditional ε-transitions, which makes the ε-closure of a set of states S (i.e., the set of states reachable from a state in S using only ε-transitions) dependent on the input symbol. Thus, we compute the ε-closure of a set of states S *with respect to the input symbol* in both the transition function and while checking if the set of states S is accepting. The transition function $\delta(S,b)$ thus first computes the ε-closure S^ε_b of S with respect to the current input symbol $b \in \mathbb{B}^k$ and then computes the set $S^\star_b$ of

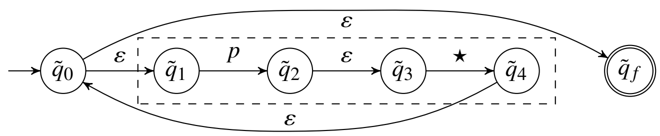


Fig. 2: The ε-NFA for $(p? \cdot \star)^*$, with the dashed rectangle showing the ε-NFA for $p? \cdot \star$

states reachable from a state in S_b^ε by following a *single* $\star$-transition. In particular, the set $S_b^\star$ is not necessarily ε-closed with respect to the next input symbol. When checking if a set of states S is accepting with respect to an input symbol $b \in \mathbb{B}^k$, we first compute the ε-closure S_b^ε of S with respect to b and then check if an accepting state is in S_b^ε. To summarize, we convert an MDL regular expression r into a DFA $\mathcal{A}_D = (Q, \mathbb{B}^k, \delta, q_0, F)$ where

- Q is the set of states of $\mathcal{A}_D$ consisting of all subsets of the set of states of $\mathcal{A}_N$;
- $\delta : Q \times \mathbb{B}^k \to Q$ is the transition function for a state relative to an input symbol;
- q_0 is the initial state of $\mathcal{A}_D$, which is a singleton consisting of the initial state of $\mathcal{A}_N$;
- $F : Q \times \mathbb{B}^k \to \mathbb{B}$ is the accepting function for a state relative to an input symbol.

We label $\mathcal{A}_N$'s nondeterministic states by $\tilde{q}$ and $\mathcal{A}_D$'s deterministic states by q.

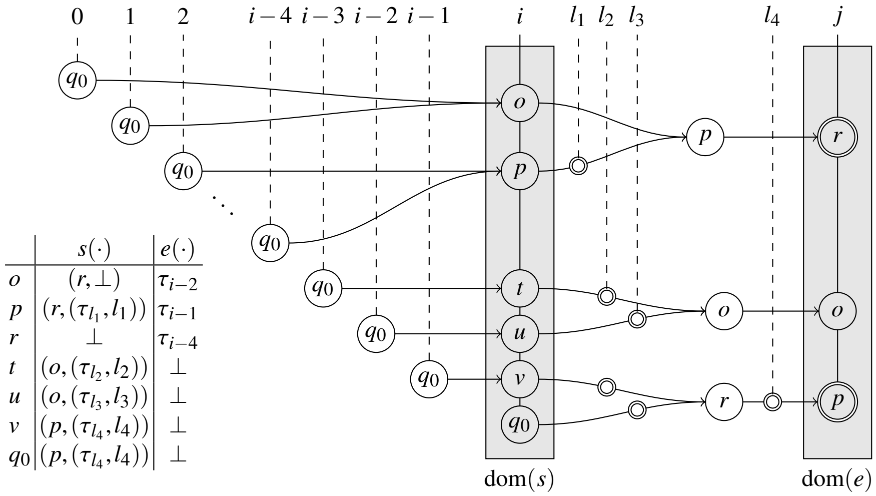
Example 2. Figure 2 shows the ε-NFA computed for the regular expression $(p? \cdot \star)^*$.

4.2 The Window Data Structure

Given a pair of time-points (i, j) with $i \leq j$, we say that the DFA $\mathcal{A}_D$ *reaches a state* q' *from a state* q *on* (i, j), denoted $q \rightsquigarrow_{(i,j)} q'$, iff the state q' is reached by running $\mathcal{A}_D$ from the state q at time-point i until time-point j. In particular, we have $q \rightsquigarrow_{(i,i)} q$, for all q and i. Furthermore, we say that $\mathcal{A}_D$ *accepts from a state* q *on* (i, j), denoted $q \rightsquigarrow_{(i,j)} \checkmark$, iff the state q' reached by $\mathcal{A}_D$ from q on (i, j) is accepting with respect to the time-point j, i.e., $F(q', b^j)$ holds. We also use the following notation: $\mathrm{dom}(f)$ of a partial function $f : X \to Y$ denotes f's domain, i.e., $\mathrm{dom}(f) = \{x \in X \mid f(x) \neq \bot\}$. For a pair $\mathit{tstp} \in \mathbb{T} \times \mathbb{N}$ of a time-stamp and time-point, $\mathit{ts}(\mathit{tstp})$ denotes the time-stamp and $\mathit{tp}(\mathit{tstp})$ the time-point.

The window data structure consists of a pair of time-points (i, j) with $i \leq j$ and two partial functions $s : Q \to Q \times ((\mathbb{T} \times \mathbb{N}) \cup \{\bot\})$ and $e : Q \to \mathbb{T}$. The function s represents the runs of $\mathcal{A}_D$ from a given state at the window's start to the state reached at the window's end and the last time-point (along with the corresponding time-stamp) within the window at which the run was in an accepting state (if such a time-point exits). The function e stores the time-stamp of the latest time-point before the window's start from which a given state at the window's end can be reached from the initial state.

Figure 3 visualizes the window data structure. Formally, the window is comprised of the table on the left. Figure 3 shows $\mathcal{A}_D$'s runs justifying the table's content. The individual runs are depicted by arrows from the initial state q_0. Whether a state is accepting depends on the current input symbol, which explains why a single state (e.g., p) may be both accepting and non-accepting at different time-points. We use standard notation for accepting states, including the smaller circles, which denote states whose name is irrelevant.



	$s(\cdot)$	$e(\cdot)$
o	$(r,\bot)$	τ_{i-2}
p	$(r,(\tau_{l_1},l_1))$	τ_{i-1}
r	$\bot$	τ_{i-4}
t	$(o,(\tau_{l_2},l_2))$	$\bot$
u	$(o,(\tau_{l_3},l_3))$	$\bot$
v	$(p,(\tau_{l_4},l_4))$	$\bot$
q_0	$(p,(\tau_{l_4},l_4))$	$\bot$

Fig. 3: The window data structure with start i and end j

The domain of s are all the states reached by running $\mathcal{A}_D$ from the initial state at a time-point before the window's start i until i (including the initial state itself obtained by running from i to i). The value of $s(q) = (q', tstp)$ for a state $q \in \mathrm{dom}(s)$ is obtained by running $\mathcal{A}_D$ further from the state q at the window's start i until the window's end j to a state q'. For example, the state r at the window's end j is reached from the states o and p at the window's start i in Figure 3. Moreover, $tstp$ is the maximum time-point after i and strictly before j such that the current state in the run from q to q' is accepting. Hence, we have $s(o) = (r,\bot)$ in Figure 3 because there is no such accepting state strictly before j in the run from the state o to r. (The state r itself, which is accepting at j, is not strictly before the window's end.) In contrast, we have $s(p) = (r,(\tau_{l_1},l_1))$ because the run from p to r contains an accepting state at time-point l_1 (which is the only accepting time-point in this run and thus also the maximum one). Similarly, we have $s(q_0) = (p,(\tau_{l_4},l_4))$ because the time-point l_4 is the maximum of the two accepting time-points in the run from the initial state q_0 at time-point i to the state p at time-point j.

The domain of e are all the states reached by running $\mathcal{A}_D$ from the initial state at a time-point strictly before the window's start i until the window's end j. The value of $e(q) = \tau$ for a state $q \in \mathrm{dom}(e)$ is the time-stamp of the maximum time-point from which q was reached from the initial state q_0. For example, $e(p) = \tau_{i-1}$ in Figure 3 because p is reached by running from q_0 at time-point $i-1$ until j. Note that p is also reached by running from i, but i is not strictly before the window's start and is thus not considered.

Formally, a window satisfies the invariant $\mathsf{window}(i, j, s, e)$ if the following holds:

- the window's start and end heads are at positions i and j;
- the domain of s, i.e., $\mathrm{dom}(s)$, are all states q such that $q_0 \rightsquigarrow_{(l,i)} q$, for some $l \leq i$;
- the domain of e, i.e., $\mathrm{dom}(e)$, are all states q such that $q_0 \rightsquigarrow_{(l,j)} q$, for some $l < i$;
- for any $q \in \mathrm{dom}(s)$: $s(q) = (q', tstp)$, where $q \rightsquigarrow_{(i,j)} q'$ and $tstp = (\tau_l, l)$ for the maximum time-point l with $i \leq l < j$ and $q \rightsquigarrow_{(i,l)} \checkmark$, or $tstp = \bot$ if no such l exists;

Trace:

τ_i	10	20	30	40
π_i	$\{p\}$	$\{\}$	$\{p\}$	$\{\}$

q	$s(\cdot)$	$e(\cdot)$		$s(\cdot)$	$e(\cdot)$		$s(\cdot)$	$e(\cdot)$		$s(\cdot)$	$e(\cdot)$
$\{\tilde{q}_0\}$	$(\{\tilde{q}_0\},\bot)$	$\bot$		$(\{\tilde{q}_4\},(10,0))$	$\bot$		$(\{\},(20,1))$	$\bot$		$(\{\},(20,1))$	$\bot$
$\{\tilde{q}_4\}$	$\bot$	$\bot$	$\xrightarrow{\mathsf{adv_e}}$	$\bot$	$\bot$	$\xrightarrow{\mathsf{adv_e}}$	$\bot$	$\bot$	$\xrightarrow{\mathsf{adv_s}}$	$(\{\},(20,1))$	$\bot$
$\{\}$	$\bot$	$\bot$		$\bot$	$\bot$		$\bot$	$\bot$		$\bot$	10
(i,j)	$(0,0)$			$(0,1)$			$(0,2)$			$(1,2)$	

Fig. 4: The trace and windows for Example 3

– for any $q \in \mathrm{dom}(e)$: $e(q) = \tau$, where $\tau = \tau_l$ is the time-stamp of the maximum time-point $l < i$ such that $q_0 \rightsquigarrow_{(l,j)} q$.

We now exemplify the window data structure by tracing its evolution through a sequence of window updates, which we manually selected. In the actual monitor, the update sequence is derived from the time-stamps in the event stream and the match operator's intervals. An update consists of advancing the window's start or end by one. We define the algorithms $\mathsf{adv_s}$ and $\mathsf{adv_e}$ that implement the window's start and end updates and their invariants in Section 4.3. Their integration into the multi-head monitors for the match operators is described in Section 4.4 and the time and space complexity of the overall monitor is analyzed in Section 4.5.

Example 3. Consider again the MDL regular expression $r = (p? \cdot \star)^*$ from Example 2 with the corresponding ε-NFA in Figure 2. We consider the trace given in Figure 4 and the sequence of window updates, where the window's end is advanced twice followed by advancing the window's start. Figure 4 depicts the window's state after initialization ($i = 0$ and $j = 0$) and after each update. Recall that a deterministic state is a subset of the non-deterministic states in Figure 2. For instance, the initial deterministic state is $q_0 = \{\tilde{q}_0\}$.

The ε-closure of $\{\tilde{q}_0\}$ at time-point 0 is $Q_0 = \{\tilde{q}_0, \tilde{q}_1, \tilde{q}_2, \tilde{q}_3, \tilde{q}_f\}$. In particular, it contains $\tilde{q}_2$ and $\tilde{q}_3$ because p is satisfied at time-point 0 and thus the conditional ε-transition from $\tilde{q}_1$ to $\tilde{q}_2$ can be taken. After advancing the window's end, the function e remains unchanged (its domain stays empty until the window's start advances). To update s, we perform a transition from $\{\tilde{q}_0\}$ at time-point 0 by following $\star$-transitions from Q_0. This way, we arrive at the next state $\{\tilde{q}_4\}$. Because Q_0 contains the accepting state $\tilde{q}_f$, the state $\{\tilde{q}_0\}$ is accepting at time-point 0. Hence, we add time-point 0 (along with the corresponding time-stamp 10) to $s(\{\tilde{q}_0\})$.

The ε-closure of $\{\tilde{q}_4\}$ at time-point 1 is $Q_1 = \{\tilde{q}_4, \tilde{q}_0, \tilde{q}_1, \tilde{q}_f\}$. In particular, it contains neither $\tilde{q}_2$ nor $\tilde{q}_3$ because the formula p is not satisfied at time-point 1 and thus the conditional ε-transition from $\tilde{q}_1$ to $\tilde{q}_2$ cannot be taken. To advance the window's end once more, no update of the function e is needed (as before). To update the function s, we perform a transition from $\{\tilde{q}_4\}$ at time-point 1 and arrive at the empty state $\{\}$ because no $\star$-transition is possible from Q_1. Because $\{\tilde{q}_4\}_1^{\varepsilon}$ contains the accepting state $\tilde{q}_f$, the state $\{\tilde{q}_4\}$ is accepting at time-point 1, and we update the time-stamp to 20 and time-point to 1 in $s(\{\tilde{q}_0\})$.

We now advance the window's start, i.e., update the window to $(1,2)$. To this end, we set $e(\{\}) = 10$ because from $s(\{\tilde{q}_0\}) = (\{\},(20,1))$ we derive that the state $\{\}$ is reached at the window's end 2 starting from the initial deterministic state $\{\tilde{q}_0\}$ at time-point 0. Next, we perform a transition (at time-point 0) from the state $\{\tilde{q}_0\}$ in $\mathrm{dom}(s)$, which yields the state $\{\tilde{q}_4\}$. Since the maximum accepting time-point 1 is within the new window $(1,2)$, we keep it and arrive at $s(\{\tilde{q}_4\}) = (\{\},(20,1))$. To compute $s(\{\tilde{q}_0\})$ for the initial deterministic state $\{\tilde{q}_0\}$, we perform two runs starting at time-point 1, one from $\{\tilde{q}_0\}$ and one from $\{\tilde{q}_4\}$, until the two states in the runs collapse or the window's end is reached. In this example, we carry out a single step and the two states collapse into $\{\}$ at time-point 2 (and the window's end is reached as well). Because time-point 1 in $s(\{\tilde{q}_4\})$ is strictly before the collapse at time-point 2, we cannot take it for $s(\{\tilde{q}_0\})$. However, since $\{\tilde{q}_0\}$ is accepting at time-point 1, we have $s(\{\tilde{q}_0\}) = (\{\},(20,1))$.

4.3 Initialization and Update of the Window Data Structure

The algorithms initializing and updating the window data structure are defined in Figure 5. The window is initialized to time-points $(0,0)$ using $\mathsf{init_w}$ (Algorithm 1), which also establishes the invariant.

Lemma 1. *The invariant* $\mathsf{window}(\mathsf{init_w})$ *holds for the initial window.*

The window (i, j, s, e) can be updated to time-points $(i, j+1)$ using the function $\mathsf{adv_e}$ (Algorithm 2).

The algorithm first updates the function e (lines 4–11). The updated domain of e is obtained by performing a transition at the window's end from all states in the original domain (line 7) and whenever two states q and q' collapse into a single state q_{new} after performing the transition, the function e associates q_{new} with the supremum of $e_{old}(q)$ and $e_{old}(q')$, using $e(q_{new})$ as an accumulator. Next $\mathsf{adv_e}$ updates the function s (lines 12–18). Its domain does not change because the window's start i remains the same. However, for any state $q \in \mathrm{dom}(s)$ with $s(q) = (q', tstp)$, a transition is performed on the state q' at the window's end (extending $q \rightsquigarrow_{(i,j)} q'$ to $q \rightsquigarrow_{(i,j+1)} q'_{new}$) and $tstp$ is updated to (τ_j, j) if $q \rightsquigarrow_{(i,j)} \checkmark$. Overall, $\mathsf{adv_e}$ preserves the window invariant.

Lemma 2. *Assume that the invariant* $\mathsf{window}(i, j, s, e)$ *holds. Then the invariant holds after advancing the window's end, i.e.,* $\mathsf{window}(\mathsf{adv_e}(i, j, s, e))$.

To advance the window's start, we must advance the domain of s and then compute $s(q_0)$ at the new window's start. We first generalize the part of the window invariant characterizing s to take into account that $s(q_0)$ might not be computed yet. To this end, we define the generalized invariant $\mathsf{svalid}(i, i', j, s)$, which asserts that s is valid for the window (i', j), but its domain contains only states reached by running from a time-point before i. In particular, $\mathsf{window}(i, j, s, e)$ implies $\mathsf{svalid}(i, i, j, s)$. Formally, $\mathsf{svalid}(i, i', j, s)$ holds if:

- $\mathrm{dom}(s)$ consists of all states q such that $q_0 \rightsquigarrow_{(l,i')} q$, for some $l \le i$;
- for any $q \in \mathrm{dom}(s)$: $s(q) = (q', tstp)$, where $q \rightsquigarrow_{(i',j)} q'$ and $tstp = (\tau_l, l)$ for the maximum time-point l with $i' \le l < j$ and $q \rightsquigarrow_{(i',l)} \checkmark$, or $tstp = \bot$ if no such l exists.

```
function init_w:
    s := (λq.⊥)
    s(q_0) := (q_0, ⊥)
    e := (λq.⊥)
    return (0, 0, s, e)
```

Algorithm 1: Initialize state

```
function adv_e(i, j, s, e):
    τ_j, b^j := read end head
    advance end head
    e_old := e
    e := (λq.⊥)
    for q ∈ dom(e_old) do
        q_new := δ(q, b^j)
        if q_new ∈ dom(e) then
            e(q_new) :=
              e(q_new) ⊔ e_old(q)
        else
            e(q_new) := e_old(q)
    for q ∈ dom(s) do
        let (q', tstp) = s(q)
        q'_new := δ(q', b^j)
        if F(q', b^j) then
            s(q) := (q'_new, (τ_j, j))
        else
            s(q) := (q'_new, tstp)
    return (i, j+1, s, e)
```

Algorithm 2: Advance end

```
function adv_d(s, i, τ_i, b^i):
    s_old := s
    s := (λq.⊥)
    for q ∈ dom(s_old) do
        let (q', tstp) = s_old(q)
        q_new := δ(q, b^i)
        if tstp ≠ ⊥ ∧ tstp = (τ_i, i) then
            s(q_new) := (q', ⊥)
        else
            s(q_new) := (q', tstp)
    return s
```

Algorithm 3: Advance dom(s)

```
function adv_s(i, j, s, e):
    τ_i, b^i := read start head
    advance start head
    let (q', tstp) = s(q_0)
    e(q') = τ_i
    s := adv_d(s, i, τ_i, b^i)
    h_cur := clone start head
    i_cur := i + 1
    q_cur := q_0
    s_cur := s
    tstp_cur := ⊥
    while i_cur < j ∧ q_cur ∉ dom(s_cur)
      do
        τ_{i_cur}, b^{i_cur} := read h_cur
        advance h_cur
        if F(q_cur, b^{i_cur}) then
            tstp_cur := (τ_{i_cur}, i_cur)
        q_cur := δ(q_cur, b^{i_cur})
        s_cur :=
          adv_d(s_cur, i_cur, τ_{i_cur}, b^{i_cur})
        i_cur := i_cur + 1
    if q_cur ∈ dom(s_cur) then
        let (q', tstp) = s_cur(q_cur)
        if tstp ≠ ⊥ then
            s(q_0) := (q', tstp)
        else
            s(q_0) := (q', tstp_cur)
    else
        s(q_0) := (q_cur, tstp_cur)
    destroy h_cur
    return (i+1, j, s, e)
```

Algorithm 4: Advance start

Fig. 5: Algorithms to initialize and update the window data structure

The auxiliary function $\mathsf{adv_d}$ (Algorithm 3) updates s by advancing time-point i' in the invariant $\mathsf{svalid}(i,i',j,s)$. To do so, we perform a transition from every state $q \in \mathrm{dom}(s_{old})$ at time-point i' to the new state $q_{new} := \delta(q, b^{i'})$ and resetting the latest accepting time-stamp time-point pair to $\bot$ if $tp(tstp) = i'$, i.e., it is no longer accepting from the new state q_{new} at $i'+1$. This function is used when advancing the domain of s from i to $i+1$ and when computing $s(q_0)$. The invariant $\mathsf{svalid}(i,i',j,s)$ is preserved by $\mathsf{adv_d}$.

Lemma 3. *Assume that the invariant* $\mathsf{svalid}(i,i',j,s)$ *holds and that* $i' < j$. *Then the invariant holds for the updated function s, i.e.,* $\mathsf{svalid}(i,i'+1,j,\mathsf{adv_d}(s,i',\tau_{i'},b^{i'}))$.

The window (i,j,s,e) with $i < j$ can be updated to the time-points $(i+1,j)$ using the function $\mathsf{adv_s}$ (Algorithm 4). This function first updates e (line 5) to account for the run $q_0 \rightsquigarrow_{(i,j)} q'$, where the state q' is obtained from the function s (line 4), which always contains the initial state q_0 in its domain.

Next $\mathsf{adv_s}$ updates s (lines 6–28). First, the domain of s is advanced by $\mathsf{adv_d}$ (line 6). This way, the invariant on s becomes $\mathsf{svalid}(i,i+1,j,s)$. To establish $\mathsf{window}(i+1,j,s,e)$, however, $\mathsf{svalid}(i+1,i+1,j,s)$ is required. Thus, it remains to compute the value of $s(q_0)$ and update s accordingly. To this end, $\mathsf{adv_s}$ performs runs from q_0 as well as from all states in $\mathrm{dom}(s)$ until the current state q_{cur} in the run from q_0 collapses with the current state of the run from a state $q \in \mathrm{dom}(s)$ or the window's end is reached (lines 12–19). The run from q_0 is simulated by updating the current state q_{cur} (initialized to q_0 on line 9). The runs from all states in $\mathrm{dom}(s)$ are simulated by updating a copy s_{cur} of the function s to $\mathsf{adv_d}(s_{cur}, i_{cur}, \tau_{i_{cur}}, b^{i_{cur}})$ at the current time-point i_{cur} of the simulation. This way, s_{cur} satisfies $\mathsf{svalid}(i, i_{cur}, j, s_{cur})$. In particular, the function s_{cur} contains the state reached at the window's end j and the latest accepting time-point on (i_{cur}, j) for all states in its domain. To account for accepting time-points on $(i+1, i_{cur})$, the algorithm also tracks the maximum accepting time-point l (represented by the pair $tstp_{cur} = (\tau_l, l) \in \mathbb{T} \times \mathbb{N}$) such that $i+1 \le l < i_{cur}$ and $q_0 \rightsquigarrow_{(i+1,l)} \checkmark$.

After the loop on lines 12–19 terminates, $\mathsf{adv_s}$ proceeds branching according to whether the current state q_{cur} collapsed with the current state of the run from a state $q \in \mathrm{dom}(s)$. If yes, then we have $q_0 \rightsquigarrow_{(i+1,i_{cur})} q_{cur}$ and also $q \rightsquigarrow_{(i+1,i_{cur})} q_{cur}$. Because the states are deterministic, the two runs from q_0 and q continue the same after i_{cur}. Hence, the run from q_{cur} at i_{cur} reaches the state q' from $s_{cur}(q_{cur}) = (q', tstp)$. If $tstp \neq \bot$, then $tstp$ represents the latest accepting time-point following q_{cur} at i_{cur} which is also the latest accepting time-point time-point pair following q_0 at $i+1$. On the other hand, if $tstp = \bot$, then there is no accepting time-point following q_{cur} at i_{cur}. Hence, the latest accepting time-point following q_0 at $i+1$ is $tstp_{cur}$. If the current state q_{cur} did not collapse with the current state of the run from any state $q \in \mathrm{dom}(s)$, then the window's end must have been reached (due to the loop condition on line 12). Then we have $i_{cur} = j$ and thus $s(q_0) = (q_{cur}, tstp_{cur})$ (line 27). Overall, $\mathsf{adv_s}$ preserves the window invariant.

Lemma 4. *Assume that the invariant* $\mathsf{window}(i,j,s,e)$ *holds and that* $i < j$. *Then the invariant holds after advancing the window's start, i.e.,* $\mathsf{window}(\mathsf{adv_s}(i,j,s,e))$.

4.4 Multi-Head Monitors for Temporal Match Operators

The algorithms implementing a step of our multi-head monitor for a past or future temporal match operator are defined using pseudocode in Figure 6.

1 **function** $\mathsf{eval_P}((a,b),(i,j,s,e))$**:**
2 $\tau_i, \_ :=$ read start head
3 $\tau_j, b^j :=$ read end head
4 **while** $i < j \wedge \tau_i + a \leq \tau_j$ **do**
5 $(i,j,s,e) := \mathsf{adv_s}(i,j,s,e)$
6 $\tau_i, \_ :=$ read start head
7 $\beta := (\exists q \in \mathrm{dom}(e).\ \tau_j \leq e(q) + b \wedge F(q, b^j)) \vee (a = 0 \wedge F(q_0, b^j))$
8 **return** $(\beta, \mathsf{adv_e}(i,j,s,e))$

Algorithm 5: Multi-head monitor's step on a formula $\langle r|_{[a,b]}$

1 **function** $\mathsf{eval_F}((a,b),(i,j,s,e))$**:**
2 $\tau_i, \_ :=$ read start head
3 $\tau_j, \_ :=$ read end head
4 **while** $\tau_j \leq \tau_i + b$ **do**
5 $(i,j,s,e) := \mathsf{adv_e}(i,j,s,e)$
6 $\tau_j, \_ :=$ read end head
7 **let** $(q', tstp) = s(q_0)$
8 $\beta := (tstp \neq \bot \wedge \tau_i + a \leq ts(tstp))$
9 **return** $(\beta, \mathsf{adv_s}(i,j,s,e))$

Algorithm 6: Multi-head monitor's step on a formula $|r\rangle_{[a,b]}$

Fig. 6: Multi-head monitor's evaluation step on a past or future match operators

To determine the Boolean verdict at a time-point j for a past match formula $\langle r|_{[a,b]}$, we must check if there exists a match from a time-point $l \leq j$ such that $\tau_j \in \tau_l + [a,b]$, i.e., $\tau_l + a \leq \tau_j \leq \tau_l + b$. Our multi-head monitor maintains a window (i,j,s,e) such that the invariant $\mathsf{window}(i,j,s,e)$ holds and $\tau_l + a \leq \tau_j$, for all $l < i$.

The algorithm $\mathsf{eval_P}$ first adjusts the window so that the time-points $l < i$ strictly before the window's start are exactly those with $l < j$ and $\tau_l + a \leq \tau_j$ (lines 4–6), i.e., it advances the window's start to the maximum $i \leq j$ such that $\tau_l + a \leq \tau_j$ for all $l < i$.

Then we seek to find a past match from a time-point $l < i$ (which implies that $\tau_l + a \leq \tau_j$) such that $\tau_j \leq \tau_l + b$. Using $\mathsf{window}(i,j,s,e)$, this amounts to checking whether there exists some $q \in \mathrm{dom}(e)$ such that $\tau_j \leq e(q) + b$ (the first disjunct on line 7). The maximality of i implies that no candidate time-point for the beginning of a past match is missed, except if $a = 0$ and the initial state is accepting at time-point j that we evaluate. The second disjunct on line 7 checks such a potential match in the interval of the form (j, j).

Lemma 1 shows that $\mathsf{window}(\mathsf{init_w})$ holds. Moreover, because $i = j = 0$ in the initial monitor's state $\mathsf{init_w}$, there exists no $l < i$, so we trivially also have $\tau_l + a \leq \tau_j$, for all $l < i$. Finally, we show that given a monitor's state satisfying these two properties, the evaluation function $\mathsf{eval_P}$ computes a sound Boolean verdict at time-point j and returns a monitor's state preserving the two properties at the next time-point $j+1$.

Lemma 5. *Assume that the invariant* $\mathsf{window}(i,j,s,e)$ *holds and* $\tau_l + a \leq \tau_j$*, for all* $l < i$*. Let* $(\beta, (i', j', s', e')) = \mathsf{eval_P}((a,b),(i,j,s,e))$*. Then, (i)* β *iff* $j \models \langle r|_{[a,b]}$*, (ii)* $j' = j + 1$*, (iii)* $\mathsf{window}(i', j', s', e')$*, and (iv)* $\tau_l + a \leq \tau_{j'}$*, for all* $l < i'$*.*

To determine the Boolean verdict at a time-point i for a future match formula $|r\rangle_{[a,b]}$, we need to check if there exists a match until a time-point $l \geq i$ such that $\tau_l \in \tau_i + [a,b]$, i.e., $\tau_i + a \leq \tau_l \leq \tau_i + b$. Our multi-head monitor maintains a window (i,j,s,e) such that the invariant $\mathsf{window}(i,j,s,e)$ holds and $\tau_l \leq \tau_i + b$, for all $i \leq l < j$.

The algorithm $\mathsf{eval_F}$ first adjusts the window so that the time-points $i \leq l < j$ are exactly those with $\tau_l \leq \tau_i + b$ for all $i \leq l < j$ (lines 4–6), i.e., it advances the window's end to the maximum j such that $\tau_l \leq \tau_i + b$ for all $i \leq l < j$.

Then the invariant $\mathsf{window}(i, j, s, e)$ implies that $q_0 \in \mathrm{dom}(s)$ and that the latest accepting time-point within the window (which coincides with the interval $[0,b]$) is stored in $s(q_0) = (q', tstp)$. It remains to check $tstp \neq \bot$ (i.e., if an accepting time-point within the window exists) and if yes, whether $ts(tstp) \geq \tau_i + a$, i.e., if the future match ends in $[a,b]$.

Lemma 1 shows that $\mathsf{window}(\mathsf{init_w})$ holds. Moreover, because $i = j = 0$ in the initial monitor's state $\mathsf{init_w}$, there exists no $i \leq l < j$, so we trivially also have $\tau_l \leq \tau_i + b$, for all $i \leq l < j$. Finally, we show that given a monitor's state satisfying these two properties, the evaluation function $\mathsf{eval_F}$ computes a sound Boolean verdict at time-point i and returns a monitor's state preserving the two properties at the next time-point $i+1$.

Lemma 6. *Assume that the invariant* $\mathsf{window}(i, j, s, e)$ *holds and* $\tau_l \leq \tau_i + b$ *for all* $i \leq l < j$. *Let* $(\beta, (i', j', s', e')) = \mathsf{eval_F}((a,b),(i, j, s, e))$. *Then, (i)* β *iff* $i \vDash |r\rangle_{[a,b]}$, *(ii)* $i' = i+1$, *(iii)* $\mathsf{window}(i', j', s', e')$, *and (iv)* $\tau_l \leq \tau_{i'} + b$ *for all* $i' \leq l < j'$.

The soundness and completeness of the overall multi-head monitor follows by induction on the structure of MDL formulas using Lemmas 5 and 6 for the cases of temporal match formulas. We denote by $\mathsf{init}(\varphi)$ the initial multi-head monitor's state for an MDL formula φ and by $\mathsf{eval}(v)$ the evaluation function of the multi-head monitor's state v (both omitted). Then, soundness and completeness amount to the following theorem.

Theorem 1. *Let* φ *be a bounded-future MDL formula,* $n \in \mathbb{N}$, *and* v *the multi-head monitor's state after applying* n *times the evaluation function* eval *starting from* $\mathsf{init}(\varphi)$. *Let* $\mathsf{eval}(v) = (v', (t, \beta))$. *Then, (i)* $t = \tau_n$ *and (ii)* β *iff* $n \vDash \varphi$.

4.5 Time and Space Complexity

By induction on the structure of MDL regular expressions, it follows that the number of states of the ε-NFA $\mathcal{A}_N$ computed by T (Figure 1) is linear in the size $|r|$ of the regular expression r. Because the set of states Q of the DFA $\mathcal{A}_D$ consists of all subsets of the set of states of $\mathcal{A}_N$, we derive $|Q| \leq 2^{O(|r|)}$. We also observe that a window can be stored in $O(|Q|)$ registers representing $\mathcal{A}_D$'s states, time-stamps, and indices into the trace.

When analyzing the time complexity, we treat the evaluation of the transition and accepting functions of the deterministic automaton $\mathcal{A}_D$ for a regular expression to be basic steps in the computation. Their precise cost depends on the actual machine model and is not analyzed here. We observe that the time complexity of initializing a temporal monitor's state and advancing the window's end is linear in the number of deterministic states, i.e., $O(|Q|)$. The time complexity of updating the window's start is $O(|Q| \cdot m)$, where m denotes the number of times a cloned reading head has been advanced during the update. Because a cloned reading head never advances beyond the window's end, it is only advanced from a time-point i_{cur} if $q_{cur} \notin \mathrm{dom}(s_{cur})$, and we have $q_{cur} \in \mathrm{dom}(s_{cur})$ next time the cloned reading head reaches i_{cur}; it follows that a cloned reading head is advanced from a time-point i_{cur} at most $|Q|$ times. Hence, the time complexity of updating the window from $(0,0)$ to (i, j) is at most $O(|Q|^2 \cdot j)$, i.e., amortized $O(|Q|^2)$ per time-point.

To bound the time and space complexity of our multi-head monitor, we first analyze how many instances of the window data structure exist at any given time. To this end, we observe that each match operator uses at most 3 reading heads over its direct subformulas, i.e., it requires at most 3 copies of the multi-head monitor for each direct subformula. Hence, the total number of temporal monitor's state instances in a multi-head monitor for an MDL formula φ is at most $3^{d(\varphi)}$, where $d(\varphi)$ is the nesting depth of the MDL formula φ. Since $d(\varphi) \leq |\varphi|$, there are at most $3^{|\varphi|}$ temporal monitor's state instances at any given time. Since an instance of a temporal monitor's state can be stored in $O(|Q|)$ registers and $|Q| \leq 2^{O(|r|)}$, the space complexity of a multi-head monitor for the formula φ is $2^{O(|\varphi|)}$ registers representing $\mathcal{A}_D$'s states, time-stamps, and indices into the trace.

To bound the time complexity of a multi-head monitor's step, we recall that the amortized time complexity of updating the window is $O(|Q|^2)$ per time-point. Because there are at most $3^{|\varphi|}$ temporal monitor's state instances at any given time, we conclude that the amortized time complexity of Algorithm 5 and 6 is $3^{|\varphi|} \cdot O(|Q|^2) \leq 2^{O(|\varphi|)}$ per time-point. Because a formula φ has $O(|\varphi|)$ subformulas, the time complexity of evaluating the entire formula can also be bounded by $2^{O(|\varphi|)}$. We summarize the complexity analysis in the following theorem.

Theorem 2. *The amortized time complexity of evaluating an MDL formula φ is at most $2^{O(|\varphi|)}$ basic steps of computation. The space complexity of storing the multi-head monitor's state for evaluating the formula φ is at most $2^{O(|\varphi|)}$ registers representing deterministic automata states, time-stamps, and indices into the trace.*

5 Implementation and Evaluation

We have implemented our multi-head MDL monitor in a tool called HYDRA(MDL), consisting of roughly 3500 lines of C++ code [21]. Our implementation mirrors the structure of the multi-head monitor presented here and consists of C++ classes for monitoring atomic predicates, Boolean operators, and temporal match operators. In fact, the implementation extends HYDRA(MTL) [20] with classes for the temporal match operators.

In addition, we have exported OCaml code from our Isabelle formalization and augmented this verified core with unverified OCaml and C code for parsing the formula and log file. We call the resulting tool VYDRA(MDL). We have used it to successfully test the correctness of HYDRA(MDL) on thousands of pseudo-random formulas and traces.

To evaluate our tools' performance, we conduct a set of experiments comparing HYDRA(MDL) and VYDRA(MDL) with HYDRA(MTL) [20], AERIAL [7], REELAY [26], R2U2 [19] and PCRE [16], a library used in many regular expression engines, e.g., `grep`. We distinguish AERIAL(MDL) that supports MDL as defined in this paper and AERIAL(MTL) that is optimized for MTL formulas. Similarly, REELAY supports past-only MTL and untimed past-only regular expressions. Moreover, time-stamps for past-only MTL are (implicitly) equal to the time-points for REELAY (in particular, they are not explicitly part of the log). R2U2 restricts the time-stamps in the same way. In addition to past-only MTL, it supports future-only MTL, but not formulas mixing past and future operators. Because we focus on MDL and interval-obliviousness, we only include REELAY and R2U2 in an experiment that demonstrates that both tools are not interval-oblivious even in the restricted setting of past-only MTL with time-stamps coinciding to time-points. Finally, PCRE supports tests similar to MDL, but restricts them to be star-free.

The time-stamps and time-points used in our algorithm are represented as 32-bit integers in HYDRA(MDL) and as arbitrary precision integers in VYDRA(MDL). The other

Experiment	Formula size	Number of formulas	Trace length	Scaling factor
IO	25	10	20 000	1–10
SZ	2–50	10	20 000	1

Fig. 7: The setup of the first two experiments

tools used in our experiments use bounded-precision machine integers as their representation. In our complexity analysis, we use an abstract model of computation, treating such values as being stored in registers that can be manipulated in a basic computation step.

We run our experiments on an Intel Core i7-8550U computer with 32 GB RAM. We measure the tools' total execution time and maximal writeable memory usage using a custom tool that performs two repetitions of each run. The tool measures the total execution time in the first repetition and calls `pmap` in a loop to determine the maximal writeable memory usage in the second repetition. Each experiment is repeated three times to minimize the impact of the execution environment. Each unfilled data point in our plots shows the average for the tool invocations with the same input parameters. We omit the negligible standard deviations. Each filled data point shows the average over a collection of a tool's data points with the same x-coordinate. We include trend lines over the filled data points in all plots. Note that the y-axis is always plotted in the logarithmic scale. Consequently, an exponential growth of a quantity looks linear and a polynomial growth looks logarithmic in the plots.

We now describe the experiments. In the first two experiments, HYDRA(MDL), VYDRA(MDL), and AERIAL are benchmarked on pseudo-random formulas and traces. In the first experiment (IO), the formulas are of a fixed size, with the time-stamp intervals of match operators scaled by a given scaling factor. In the second experiment (SZ), the formulas grow in size with small bounds in the intervals of match operators. In both experiments, the traces are of a fixed size. The parameters of the first two experiments are summarized in Figure 7.

The pseudo-random formulas are produced by mutually recursive generators for formula and regular expressions for a predefined size and maximum interval bounds. A formula φ of size $s > 0$ is generated as follows: (i) if $s = 1$, then $\varphi = p$, for an atomic predicate $p \in \{p_0, \ldots, p_{15}\}$ chosen uniformly at random; (ii) if $s = 2$, a top-level unary operator op is selected uniformly at random among the three unary operators; and (iii) if $s \geq 3$, a top-level operator op is selected uniformly at random among the recursive MDL operators. A regular expression r of size $s > 0$ is generated as follows: (i) if $s = 1$, then $r = \star$; (ii) if $s = 2$, a top-level operator op is selected uniformly at random among $\star$ and $(\cdot)^*$; and (iii) if $s \geq 3$, a top-level operator op is selected uniformly at random among all regular expression operators. If the top-level operator op of a formula has an interval, then the interval is generated as follows: (i) with probability $\frac{1}{4}$, the interval $[0,0]$ is chosen; (ii) with probability $\frac{1}{4}$, an interval $[0,r]$ is chosen with r distributed uniformly in $\{1, \ldots, \Delta\}$, or $\{1, \ldots, \Delta\} \cup \{\infty\}$, for a predefined Δ (in our experiments, we use $\Delta = 16$); and (iii) with probability $\frac{1}{2}$, an interval $[l,r]$ is chosen with $l \in \{1, \ldots, \Delta\}$ and $r \in \{l, \ldots, \Delta\}$, or $r \in \{l, \ldots, \Delta\} \cup \{\infty\}$, distributed uniformly at random. Finally, if the top-level operator op is unary, a subformula φ (or subexpression r) of size $s - 1$ is generated recursively.

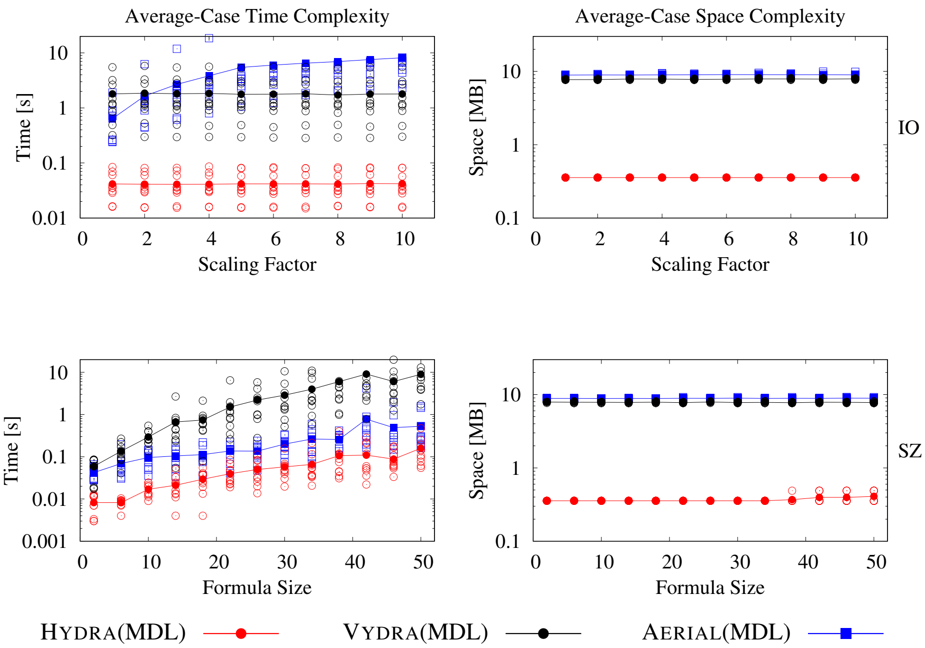


Fig. 8: Evaluation results for the randomized experiments IO and SZ

If op is binary, two random subformulas (or subexpressions) of sizes s_1 and $s-1-s_1$ are generated recursively, where $s_1 \in \{1, \ldots, s-2\}$ is chosen uniformly at random.

The pseudo-random traces are produced by a generator for a predefined event rate *er* [1]. Each trace contains events with 2 000 different time-stamps. The time-stamp differences are distributed uniformly in $\{1, \ldots, \varDelta\}$, for a predefined $\varDelta$; in our experiments, we use $\varDelta = 4$. The atomic predicates are generated as follows: (i) independently with probability $1 - \frac{1}{\varDelta \cdot er}$, an atomic predicate $p_0, \ldots, p_3$ is included; and (ii) independently with probability $\frac{1}{2}$, an atomic predicate $p_4, \ldots, p_{15}$ is included.

Figure 8 summarizes the results for the experiments IO and SZ. The experiment IO shows that neither the time nor space complexity of HYDRA(MDL) and VYDRA(MDL) depends on the numerical values in the intervals, i.e., both tools are interval-oblivious. AERIAL(MDL)'s time complexity grows with increasing interval bounds because the algorithm works with formulas whose intervals are shifted by offsets up to the numerical bounds in the intervals [1]. Similarly, AERIAL(MDL)'s space complexity grows with increasing interval bounds, but it is dominated by the constant overhead of the runtime environment before AERIAL(MDL) times out. The experiment SZ shows that HYDRA(MDL) outperforms AERIAL(MDL) also when increasing the formulas' size.

The worst-case experiment (WC) reported in our previous work [20] results in space complexity of online monitoring that is exponential in the formula size (in fact, already to produce a single Boolean verdict for the first time-point of a repetition). It is conducted

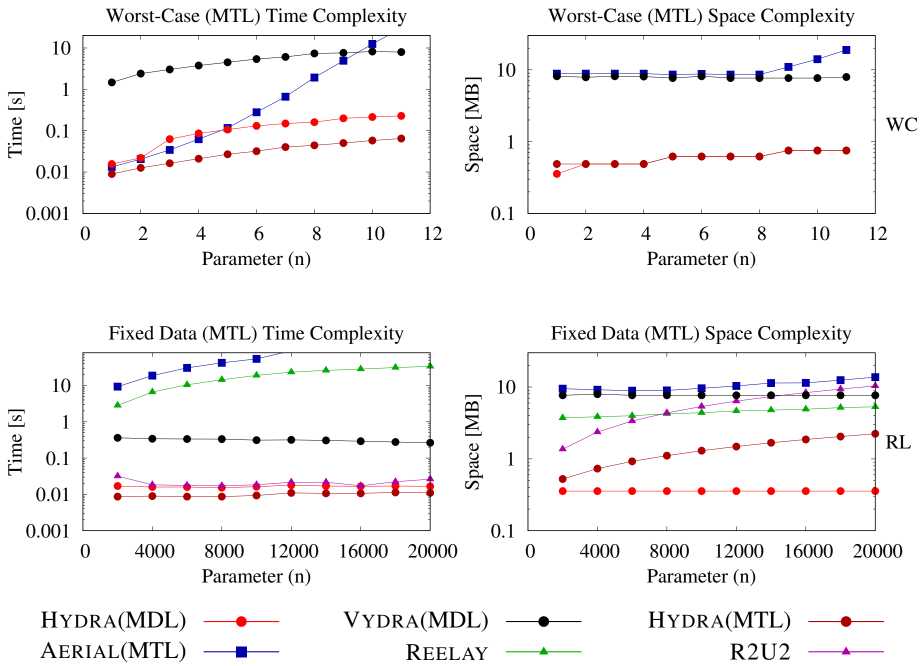


Fig. 9: Evaluation results for the experiment WC and RL

on formulas of the form:

$$\Phi_n = \bigcirc_{[1,1]}(\neg e \,\mathsf{U}_{[0,0]}\, (\neg e \wedge \bigwedge_{i=1}^{n}(p_i \Rightarrow \Box_{[0,0]}(e \Rightarrow p_i)) \wedge \bigwedge_{i=1}^{n}(\neg p_i \Rightarrow \Box_{[0,0]}(e \Rightarrow \neg p_i)))).$$

A trace, parameterized by $n \in \mathbb{N}$, is constructed by repeating the following pattern and increasing the initial time-stamp of each repetition so that they are independent: the first event is an empty event with a time-stamp τ; then for each subset $X \in \mathcal{X} \subseteq 2^{P^{\Phi_n} \setminus \{e\}}$ of atomic predicates without e, we include an event with the atomic predicates X and a time-stamp $\tau + 1$. Finally, for some $X \subseteq P^{\Phi_n}$, we include an event with the atomic predicates $X \cup \{e\}$ and a time-stamp $\tau + 1$. Figure 9 summarizes the evaluation results. We observe that HYDRA(MDL)'s and VYDRA(MDL)'s time complexity is polynomial, whereas AERIAL(MDL)'s is exponential. (Recall that all y-axes are in logarithmic scale.) HYDRA(MTL) is the fastest here, as it is optimized for the more restricted logic.

The REELAY comparison experiment (RL) is conducted on formulas and traces described by Ulus [25]. The formulas are of the form: $\mathsf{DELAY}(n) = p\,\mathsf{S}_{[n,n]}\,q$. A trace, parameterized by $n \in \mathbb{N}$, is constructed with p being always true and q being true at every other time-point (with time-stamps being equal to time-points). Figure 9 summarizes the results for this experiment. It confirms that the time complexity of both AERIAL(MTL) and REELAY grows when increasing n, i.e., neither of these tools is interval-oblivious. For AERIAL(MTL), the reason is again that the algorithm considers all interval-shifted formulas. The algorithm implemented in REELAY combines interval-shifted formulas

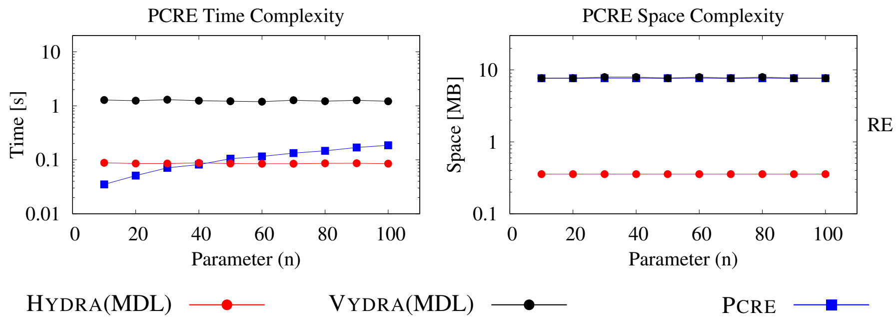


Fig. 10: Evaluation results for the experiment RE

with consecutive offsets. Nevertheless, the event pattern in the log files used in the experiment prevents this optimization and shows that REELAY's time and space complexity still depends on the interval bounds in the worst-case. Also, R2U2's space complexity depends on the interval bounds. Its time complexity is comparable to HYDRA(MDL)'s on this simple formula. In contrast, the time complexity of HYDRA(MTL), HYDRA(MDL), and VYDRA(MDL) is confirmed to be independent of the parameter n. Finally, the experiment shows that HYDRA(MTL)'s space complexity is not interval-oblivious.

The PCRE comparison experiment (RE) is conducted on formulas of the form: $\Psi_n = \langle (a? \cdot \star \cdot b? \cdot \star)^* |_{[2n,2n]}$, which correspond to $r_n =$ `(?<=(ab){n}).` using the syntax of Perl compatible regular expressions. We point out that *lookbehinds* do not consume matched symbols and thus produce overlapping matches (just like in MDL). The text in which the regular expressions r_n are searched consists of 10^5 occurrences of the pattern *ab*, i.e., a total of $2 \cdot 10^5$ symbols. For HYDRA(MDL) and VYDRA(MDL), this text is encoded into a log whose events correspond to the text's symbols. Thus, the log also consists of $2 \cdot 10^6$ events. The log's time-stamps are consecutive integers denoting the number of symbols up to the respective position. The evaluation results are summarized in Figure 10. Because PCRE starts a new search for matching `(ab){n}` at each position in the text, its time complexity grows linearly in the parameter n. In contrast, HYDRA(MDL)'s and VYDRA(MDL)'s time complexity does not depend on n, as the parameter n only occurs in the interval bounds of Ψ_n.

HYDRA(MDL) outperforms all other tools by an order of magnitude in all experiments with respect to memory requirements (except for HYDRA(MTL) in the WC experiment, which is on par with HYDRA(MDL)).

6 Conclusion

We presented a new monitoring algorithm for metric dynamic logic (MDL) that follows the multi-head paradigm. Our monitor is the first event-rate independent (assuming registers) monitor for MDL that produces a stream of Boolean verdicts. This is a significant improvement over the event-rate independent monitor AERIAL in terms of the monitor's interface: Boolean verdicts are much easier for humans to understand than AERIAL's non-standard equivalence verdicts. Additionally, our monitor is interval-oblivious: The constants occurring in the formula's metric constraints have no impact on the monitor's time- and memory consumption. To our knowledge, this property is unprecedented for monitors for metric specification languages in the point-based setting.

Our algorithm may, however, require exponentially many heads in the monitored formula's size. This exponential dependence seems daunting in theory, but it seems to be unproblematic in practice. We have validated this claim by implementing our algorithm in the HYDRA(MDL) tool and evaluating its performance in a series of case studies. For example, HYDRA(MDL) can process randomly generated MDL formulas with 50 operators on traces with 20 000 events in about 100 milliseconds on average.

Acknowledgments. This research is supported by the Swiss National Science Foundation grant "Big Data Monitoring" (167162). We thank the anonymous reviewers for detailed comments.

References

1. Basin, D., Bhatt, B., Krstić, S., Traytel, D.: Almost event-rate independent monitoring. Form. Methods Syst. Des. **54**(3), 449–478 (2019)
2. Basin, D., Caronni, G., Ereth, S., Harvan, M., Klaedtke, F., Mantel, H.: Scalable offline monitoring of temporal specifications. Form. Methods Syst. Des. **49**(1-2), 75–108 (2016)
3. Basin, D., Harvan, M., Klaedtke, F., Zalinescu, E.: Monitoring data usage in distributed systems. IEEE Trans. Software Eng. **39**(10), 1403–1426 (2013)
4. Basin, D., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15 (2015)
5. Basin, D., Klaedtke, F., Zalinescu, E.: Algorithms for monitoring real-time properties. In: RV 2011. LNCS, vol. 7186, pp. 260–275. Springer (2011)
6. Basin, D., Klaedtke, F., Zalinescu, E.: The MonPoly monitoring tool. In: RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 19–28. EasyChair (2017)
7. Basin, D., Krstic, S., Traytel, D.: AERIAL: almost event-rate independent algorithms for monitoring metric regular properties. In: RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 29–36. EasyChair (2017)
8. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: TeSSLa: Temporal stream-based specification language. In: SBMF 2018. LNCS, vol. 11254, pp. 144–162. Springer (2018)
9. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: TIME 2005. pp. 166–174. IEEE Computer Society (2005)
10. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: IJCAI 2013. pp. 854–860. IJCAI/AAAI (2013)
11. Faymonville, P., Finkbeiner, B., Schledjewski, M., Schwenger, M., Stenger, M., Tentrup, L., Torfah, H.: StreamLAB: Stream-based monitoring of cyber-physical systems. In: CAV 2019. LNCS, vol. 11561, pp. 421–431. Springer (2019)
12. Gorostiaga, F., Sánchez, C.: Striver: Stream runtime verification for real-time event-streams. In: RV 2018. LNCS, vol. 11237, pp. 282–298. Springer (2018)
13. Havelund, K., Peled, D., Ulus, D.: First order temporal logic monitoring with BDDs. In: FMCAD 2017. pp. 116–123. IEEE (2017)
14. Havelund, K., Peled, D., Ulus, D.: DejaVu: A monitoring tool for first-order temporal logic. In: MT@CPSWeek 2018. pp. 12–13. IEEE (2018)

15. Havelund, K., Reger, G., Thoma, D., Zalinescu, E.: Monitoring events that carry data. In: Lectures on Runtime Verification, LNCS, vol. 10457, pp. 61–102. Springer (2018)
16. Hazel, P.: PCRE - Perl Compatible Regular Expressions. `https://www.pcre.org/` (2018)
17. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. **2**(4), 255–299 (1990)
18. Mamouras, K., Raghothaman, M., Alur, R., Ives, Z.G., Khanna, S.: StreamQRE: modular specification and efficient evaluation of quantitative queries over streaming data. In: PLDI 2017. pp. 693–708. ACM (2017)
19. Moosbrugger, P., Rozier, K.Y., Schumann, J.: R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. Form. Methods Syst. Des. **51**(1), 31–61 (2017)
20. Raszyk, M., Basin, D., Krstić, S., Traytel, D.: Multi-head monitoring of metric temporal logic. In: ATVA 2019. LNCS, vol. 11781, pp. 133–150. Springer (2019)
21. Raszyk, M., Basin, D., Traytel, D.: Formalization, implementation, and evaluation associated with this paper., `https://bitbucket.org/krle/hydra/downloads/hydra-mdl.zip`
22. Raszyk, M., Basin, D., Traytel, D.: From nondeterministic to multi-head deterministic finite-state transducers. In: ICALP 2019. LIPIcs, vol. 132, pp. 127:1–127:14. Schloss Dagstuhl—Leibniz-Zentrum für Informatik (2019)
23. Raszyk, M., Basin, D., Traytel, D.: Multi-head monitoring of metric dynamic logic (extended report) (2020), `https://bitbucket.org/krle/hydra/src/master/HYDRA_MDL.pdf`
24. Sánchez, C.: Online and offline stream runtime verification of synchronous systems. In: RV 2018. vol. 11237, pp. 138–163. Springer (2018)
25. Ulus, D.: Online monitoring of metric temporal logic using sequential networks. CoRR **abs/1901.00175** (2019), `http://arxiv.org/abs/1901.00175`
26. Ulus, D.: REELAY. `https://github.com/doganulus/reelay-codegen` (2019)