

ONLINE MONITORING OF METRIC TEMPORAL LOGIC USING SEQUENTIAL NETWORKS

DOGAN ULUS 

Boğaziçi University, Istanbul, Türkiye
e-mail address: dogan.ulus@bogazici.edu.tr

ABSTRACT. Metric Temporal Logic (**MTL**) is a popular formalism to specify temporal patterns with timing constraints over the behavior of cyber-physical systems with application areas ranging in property-based testing, robotics, optimization, and learning. This paper focuses on the unified construction of sequential networks from **MTL** specifications over discrete and dense time behaviors to provide an efficient and scalable online monitoring framework. Our core technique, future temporal marking, utilizes interval-based symbolic representations of future discrete and dense timelines. Building upon this, we develop efficient update and output functions for sequential network nodes for timed temporal operations. Finally, we extensively test and compare our proposed technique with existing approaches and runtime verification tools. Results highlight the performance and scalability advantages of our monitoring approach and sequential networks.

1. INTRODUCTION

Monitoring temporal behaviors of complex engineered systems during their execution has important application areas ranging from system verification and anomaly detection to supervisory control. As modern computing systems grow increasingly complex, requiring highly interactive and sophisticated features, the need for effective and more versatile monitoring solutions increases. Such high levels of complexity necessitate runtime monitoring of a vast array of temporal properties, including timing constraints and adherence to well-defined behavioral patterns. Beyond functionality, the concern for performance is also crucial in any runtime monitoring activity, given the inherent overhead introduced to these systems. Therefore, fast and versatile runtime monitoring solutions are important assets for ensuring the correctness and smooth operation of these complex systems.

It is often desirable to construct efficient runtime monitors automatically from high-level declarative specifications that describe the system behavior in an unambiguous language. Initially proposed for formal verification, Linear-time Temporal Logic (**LTL**) [Pnu77] and its timed extensions, such as Metric Temporal Logic (**MTL**) [Koy90], have become popular formalisms in academia and industry to specify the temporal behavior of real-time reactive systems. These formalisms have found diverse application areas in robotics, optimization, and property-based testing [BDD⁺18, SSA⁺19].

Key words and phrases: runtime verification, temporal logic, sequential networks, real-time systems, cyber-physical systems.

This work presents a unified approach for constructing runtime monitors over discrete and dense time behaviors from **LTL** and **MTL** specifications. Leveraging algebraic sequential networks, our proposed approach simplifies the transition from untimed to timed specifications and achieves efficient and scalable runtime monitor construction across both time models. We study and emphasize the structural advantages of sequential networks over finite automata, another general solution for runtime monitor construction. Unlike automata-based constructions, sequential networks offer substantial benefits in compositionality, extensibility, implementability, and scalability, all while preserving functional equivalence in Boolean contexts. Consequently, these structural advantages position sequential networks as a powerful and adaptable model of computation for tackling diverse runtime monitoring tasks within complex real-time systems.

Sequential networks in this paper are directly constructed from the past fragments of **LTL** and **MTL**. The restriction to the past temporal connectives is twofold: (1) Future-oriented (acausal) monitoring is inherently more expensive than past-oriented (causal) monitoring. The worst-case exponential cost of bookkeeping, among all possibilities in the future, cannot be avoided unless the output at time t is delayed by some duration d depending on the formula. (2) However, the practical value of delaying seems nonexistent for a truly online/reactive setting as we need an output from the monitor at the current time t rather than the time $t + d$, which may be too late. This is especially important when the monitor's output is used to make a timely decision, as in the supervisory/reactive control systems. Consequently, we consider future temporal operators to be a costly feature that does not offer significant practical benefits in online monitoring applications, and we restrict ourselves to the past fragment for online monitoring applications in this paper. The basic technique behind constructing sequential networks from past temporal logic formulas involves associating each subformula with a state variable to store relevant information and subsequently updating/manipulating them at each time step. This paper introduces the *future temporal marking technique* as a novel and effective approach to handling timed operators for sequential network constructions from **PastMTL** over discrete and dense time temporal behaviors. We propose a novel set of update and output equations for each timed operator to mark future time intervals according to timing constraints in the formula. Notably, our dense time construction seamlessly extends the discrete time construction, avoiding the need for a complete overhaul or reliance on naive discretization as in earlier attempts. This facilitates smooth transitions between untimed and timed specifications, as well as between discrete and dense time models, paving the way for a unified treatment of temporal operators across diverse applications and use cases. As a result, our approach enables the construction of simpler and more extensible monitors, leading to improved efficiency and adaptability for a wider range of temporal logic monitoring tasks.

The structure of the paper is as follows. Section 2 is dedicated to definitions of sequential networks and temporal logic over discrete and dense time behaviors. Section 3 and Section 4 present and explain our discrete time and dense time sequential network constructions from **PastMTL** specifications. Section 5 presents our online discrete and dense time monitoring framework, **Reelay**¹, as well as the evaluation and benchmark results. We then summarize related work in Section 6 before concluding with a summary of our contributions and promising avenues for future research in Section 7.

¹<https://github.com/doganulus/reelay>

2. PRELIMINARY DEFINITIONS AND BACKGROUND

In this paper, we use both discrete and dense time domains, denoted as \mathbb{T} in general. Specifically, we consider the set \mathbb{N} of non-negative integers and the set $\mathbb{Q}_{>0}$ of positive rational numbers as our discrete and dense time domains, respectively.

Sequential Networks. The online monitoring task can be naturally described as a sequence transformation [Ran58] from temporal behaviors to monitor verdicts for each time step. A sequential network is an abstract machine that consists of a finite set \mathcal{C} of computation nodes and implements a sequence-to-sequence transformation, which yields an output sequence $y_1 y_2 \dots y_k \dots$ from an input sequence $\mathbf{X}_1 \mathbf{X}_2 \dots \mathbf{X}_k \dots$ given. Each node $c \in \mathcal{C}$ is associated with an update function V_c and an output function Y_c . These node functions contribute to the network's overall update function V and output function Y . The state valuation vector \mathbf{V}_k collectively represents the internal state of network nodes at time step k , and \mathbf{V}_0 is called the initial state valuation vector. At each time step k , the network updates its state vector \mathbf{V}_k by calling node update functions and yielding the current output value y_k with respect to the previous state vector \mathbf{V}_{k-1} and the current input vector \mathbf{X}_k . We then completely characterize a sequential network by the following elements:

- (1) The initial state valuation vector \mathbf{V}_0 of length $|\mathcal{C}|$
- (2) The node update functions $V_c : \mathbf{V}_k(c) \Leftarrow V_c(\mathbf{V}_{k-1}, \mathbf{X}_k)$ for each node $c \in \mathcal{C}$
- (3) The network output function $Y : \mathbf{Y}_k \Leftarrow Y(\mathbf{V}_{k-1}, \mathbf{X}_k)$

Data types of inputs, outputs, and state variables of the network may vary across applications. The simplest class of sequential networks, in which all data types are Booleans, is called Boolean sequential networks or digital sequential circuits when realized using Boolean logic gates and memory elements. The class of Boolean sequential networks precisely recognizes regular languages and is thus functionally equivalent to finite automata. The term algebraic sequential networks denotes generalizations of Boolean sequential networks into other compatible algebraic structures, such as the algebra of sets.

Past Linear-time Temporal Logic. Past Linear-time Temporal Logic (**PastLTL**) extends the propositional logic with past temporal modalities such as the *Previously* (\mathbf{Y}), *past Always* (\mathbf{H}), *past Eventually* (\mathbf{P}), and *Since* (\mathbf{S}). **PastLTL** provides a robust framework for expressing and reasoning about temporal ordering between system states.

Given a finite set P of atomic predicates, the formulas of **PastLTL** are inductively built using the following grammar:

$$\varphi := \top \mid \perp \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \mathbf{S} \varphi_2$$

where $p \in P$. The truth of an arbitrary **PastLTL** formula φ at a given time instant t over an arbitrary discrete time behavior w , denoted as $(w, t) \models \varphi$, is defined inductively in the following. First, the following statements define the propositional fragment of **PastLTL**.

$$\begin{array}{lll} (w, t) \models \top & \leftrightarrow & \mathbf{T} \\ (w, t) \models \perp & \leftrightarrow & \mathbf{F} \\ (w, t) \models p & \leftrightarrow & w_p(t) = \mathbf{T} \\ (w, t) \models \neg\varphi & \leftrightarrow & (w, t) \not\models \varphi \\ (w, t) \models \varphi_1 \wedge \varphi_2 & \leftrightarrow & (w, t) \models \varphi_1 \text{ and } (w, t) \models \varphi_2 \end{array} \quad (2.1)$$

It is clear how to extend the definitions of negation (\neg) and conjunction (\wedge) of the other Boolean operators of disjunction (\vee), implication (\rightarrow), double implication (\leftrightarrow), etc.

In the literature, temporal modalities can have reflexive and irreflexive versions, which differ in how they handle the current time point. The distinction between these two versions becomes significant when discussing the expressive power. First, the following statement defines the irreflexive version of the *Since* modality:

$$(w, t) \models \varphi_1 S \varphi_2 \leftrightarrow \exists t' < t. (w, t') \models \varphi_2 \text{ and } \forall t' < t'' < t. (w, t'') \models \varphi_1 \quad (2.2)$$

Other commonly used past temporal modalities *Previously* (Y), *past Eventually* (P), and *past Always* (H) can be derived from the *Since* modality using equivalences

$$(a) Y \varphi \equiv \perp S \varphi, \quad (b) P \varphi \equiv \top S \varphi, \quad \text{and} \quad (c) H \varphi \equiv \neg P \neg \varphi. \quad (2.3)$$

Then we can add the following statements to the satisfaction relation:

$$(w, t) \models Y \varphi \leftrightarrow (w, t-1) \models \varphi \quad (2.4)$$

$$(w, t) \models P \varphi \leftrightarrow \exists t' < t. (w, t') \models \varphi \quad (2.5)$$

$$(w, t) \models H \varphi \leftrightarrow \forall t' < t. (w, t') \models \varphi \quad (2.6)$$

Observe that the irreflexive versions of temporal modalities exclude the current time point t from their range of quantification. On the other hand, the reflexive versions differ from their irreflexive counterparts by additionally considering the current time point. The following statement defines the reflexive version of the *Since* modality:

$$(w, t) \models \varphi_1 S' \varphi_2 \leftrightarrow \exists t' \leq t. (w, t') \models \varphi_2 \text{ and } \forall t' < t'' \leq t. (w, t'') \models \varphi_1 \quad (2.7)$$

From this definition, we can derive reflexive versions of the *Past Eventually* and the *Past Always* operators similarly:

$$(w, t) \models P' \varphi \leftrightarrow \exists t' \leq t. (w, t') \models \varphi \quad (2.8)$$

$$(w, t) \models H' \varphi \leftrightarrow \forall t' \leq t. (w, t') \models \varphi \quad (2.9)$$

Yet it is well known that the reflexive version of the *Since* modality is expressively weaker than the irreflexive one as it cannot express the *Previously* modality using Equivalence 2.3(a) or another way. Therefore, the *Previously* modality must be added explicitly to the grammar when using reflexive definitions to preserve expressiveness.

Past Metric Temporal Logic. Past Metric Temporal Logic (PastMTL) extends PastLTL with a timed variant of the *Since* modality, denoted by $S_{[a:b]}$, where a and b denote lower and upper bounds restricting the range of quantification for the modality. We omit a and b in the notation if there is no constraint on the lower and upper end, respectively. We refer to a temporal modality untimed if there is no constraint on both ends. The following statement defines the timed and irreflexive version of the *Since* modality:

$$(w, t) \models \varphi_1 S_{[a:b]} \varphi_2 \leftrightarrow \exists t' < t. (w, t') \models \varphi_2 \text{ and} \\ \forall t' < t'' < t. (w, t'') \models \varphi_1 \text{ and} \\ t - b \leq t' < t - a \quad (2.10)$$

From this definition, timed versions of *Past Eventually* and *Past Always* operators are derived using Equivalences 2.3(b) and 2.3(c), respectively, as follows.

$$(w, t) \models P_{[a:b]} \varphi \leftrightarrow \exists t' < t. (w, t') \models \varphi \text{ and } t - b \leq t' < t - a \quad (2.11)$$

$$(w, t) \models H_{[a:b]} \varphi \leftrightarrow \forall t' < t. (w, t') \models \varphi \text{ and } t - b \leq t' < t - a \quad (2.12)$$

Notice that the formula $\perp S \varphi$ is equivalent to \perp for any formula φ under dense time interpretation, unlike the discrete case. Hence it is meaningless to define the *Previously* modality as one should expect in dense time.

However, when we interpret MTL over discrete time behaviors, we can always replace timed irreflexive *Since* connective with the *Previously* connective as defined in Equation 2.4 and timed reflexive *Since* defined as follows:

$$(w, t) \models \varphi_1 S'_{[a:b]} \varphi_2 \leftrightarrow \exists t' \leq t. (w, t') \models \varphi_2 \text{ and} \\ \forall t' < t'' \leq t. (w, t'') \models \varphi_1 \text{ and} \\ t - b \leq t' \leq t - a \quad (2.13)$$

From this definition, we can derive reflexive versions of the timed Past Eventually and the timed Past Always operators similarly:

$$(w, t) \models P'_{[a:b]} \varphi \leftrightarrow \exists t' \leq t. (w, t') \models \varphi \text{ and } t - b \leq t' \leq t - a \quad (2.14)$$

$$(w, t) \models H'_{[a:b]} \varphi \leftrightarrow \forall t' \leq t. (w, t') \models \varphi \text{ and } t - b \leq t' \leq t - a \quad (2.15)$$

This paper adopts a careful and consistent approach to applying reflexive and irreflexive semantics of temporal modalities. Specifically, we exclusively use reflexive semantics (Eq. 2.4, 2.7 and 2.13) of temporal modalities when interpreting formulas over discrete time behaviors. On the other hand, we exclusively use irreflexive semantics (Eq. 2.2 and 2.10) when interpreting formulas over dense time behaviors. In this manner, our objective is to establish well-behaving default presets for online temporal logic monitoring applications. Once this is understood, we drop the prime notation for reflexive modalities and use the same notation for temporal modalities across discrete and dense time settings.

3. DISCRETE TIME SEQUENTIAL NETWORK CONSTRUCTIONS

This section describes how to construct sequential networks from *PastLTL* and *PastMTL* specifications for discrete time behaviors. We assume that we observe a finite set of propositions at each discrete time point, which can be represented as a Boolean vector. The sequential network is fed with these vectors incrementally, one at a time. To track the passage of time for timed properties, we use a global time counter that increments with each discrete time step, corresponding to the sequence index. Finally, it's important to remember that in the discrete setting, we always use reflexive semantics for temporal modalities and specify timing constraints over integers.

3.1. Sequential Networks from *PastLTL*. The *PastLTL* monitor construction presented here serves as a crucial preparation step for constructing monitors from timed specifications in the following sections. This construction shares significant similarities with the dynamic programming technique described in [HR04] while seemingly distinct in terminology and presentation. Yet these differences and changing the point of view are important in extending the approach toward metric extensions.

Given a **PastLTL** formula, we construct a sequential network consisting of nodes representing each subformula of the formula. Each node maintains a Boolean state variable, collectively forming a state valuation vector \mathbf{V} . Initially, all nodes are set to false, such that $\mathbf{V}_0 = \perp$. At each time point k , the network's update equations determine the new state valuation of each node based on its operator type and the current valuations of other nodes. These update equations defined based on the operator type are as follows:

$$\begin{aligned}
 \mathbf{V}_k(p) &\iff \mathbf{X}_k(p) := \begin{cases} \mathbf{T} & \text{if the proposition } p \text{ holds at time point } k \\ \mathbf{F} & \text{otherwise.} \end{cases} \\
 \mathbf{V}_k(\neg\varphi) &\iff \neg\mathbf{Y}_k(\varphi) \\
 \mathbf{V}_k(\varphi_1 \wedge \varphi_2) &\iff \mathbf{Y}_k(\varphi_1) \wedge \mathbf{Y}_k(\varphi_2) \\
 \mathbf{V}_k(\varphi_1 \vee \varphi_2) &\iff \mathbf{Y}_k(\varphi_1) \vee \mathbf{Y}_k(\varphi_2) \\
 \mathbf{V}_k(\mathbf{Y} \varphi) &\iff \mathbf{Y}_{k-1}(\varphi) \\
 \mathbf{V}_k(\mathbf{P} \varphi) &\iff \mathbf{Y}_k(\varphi) \vee \mathbf{V}_{k-1}(\mathbf{P} \varphi) \\
 \mathbf{V}_k(\mathbf{H} \varphi) &\iff \mathbf{Y}_k(\varphi) \wedge \mathbf{V}_{k-1}(\mathbf{H} \varphi) \\
 \mathbf{V}_k(\varphi_1 \mathbf{S} \varphi_2) &\iff \mathbf{Y}_k(\varphi_2) \vee (\mathbf{Y}_k(\varphi_1) \wedge \mathbf{V}_{k-1}(\varphi_1 \mathbf{S} \varphi_2))
 \end{aligned} \tag{3.1}$$

where \mathbf{X}_k is the propositional input vector. The output function $\mathbf{Y}_k(\varphi) \iff \mathbf{V}_k(\varphi)$ determines the output at the time point k . Since the output function is trivial for the untimed case, it is ignored in monitor constructions from untimed specifications like [HR04, HPU17]. However, we must distinguish the state and output values of network nodes when dealing with timed extensions in the following.

We now illustrate the compositionality of sequential network construction with an example. Consider a **PastLTL** formula $\varphi := (p \vee q) \mathbf{S} \neg r$, which contains three propositions and three non-leaf subformulas. We then construct a sequential network from φ , which has six nodes (all initialized to false) with update equations and the output function given in Table 1. Easily seen, we can construct the same sequential network from previously constructed monitors of $(p \vee q)$ and $\neg r$. Given two monitors for **PastLTL** formulas $\varphi_1 := (p \vee q)$ and $\varphi_2 := \neg r$, we can obtain a new monitor for $\varphi_1 \mathbf{S} \varphi_2$ by joining state variables from both monitors plus adding a new state variable for the (topmost) Since operator. Then, the outputs of φ_1 and φ_2 monitors become arguments of the update equation of the new state. In other words, output equations of φ_1 and φ_2 are embedded into the update equation of

$\mathbf{V}_k(p) : \text{bool}$	\iff	$\mathbf{X}_k(p) : \text{bool}$
$\mathbf{V}_k(q) : \text{bool}$	\iff	$\mathbf{X}_k(q) : \text{bool}$
$\mathbf{V}_k(r) : \text{bool}$	\iff	$\mathbf{X}_k(r) : \text{bool}$
$\mathbf{V}_k(\neg r) : \text{bool}$	\iff	$\neg\mathbf{Y}_k(r) : \text{bool}$
$\mathbf{V}_k(p \vee q) : \text{bool}$	\iff	$\mathbf{Y}_k(p) : \text{bool} \vee \mathbf{Y}_k(q) : \text{bool}$
$\mathbf{V}_k(\varphi) : \text{bool}$	\iff	$\mathbf{Y}_k(\neg r) : \text{bool} \vee (\mathbf{Y}_k(p \vee q) : \text{bool} \wedge \mathbf{V}_{k-1}(\varphi) : \text{bool})$
$\mathbf{Y}_k(\varphi) : \text{bool}$	\iff	$\mathbf{V}_k(\varphi) : \text{bool}$

TABLE 1. Sequential network constructed from the LTL formula $\varphi := (p \vee q) \mathbf{S} \neg r$. Equations are annotated by type information for the state and input variables, which are always Boolean for networks constructed from **PastLTL** specifications.

$\varphi_1 \mathbf{S} \varphi_2$ at the composition. Note that the update order for state variables is critical as we use reflexive semantics. Therefore, the implementation must ensure subformulas are updated earlier than their parents. This is always possible for LTL formulas due to the acyclic nature of LTL parse trees, which allows for methods like post-order traversal or topological sorting to establish a well-defined evaluation order.

3.2. Sequential Networks from PastMTL. This section extends the PastLTL monitor construction in Section 3.1 towards PastMTL specifications. Timed specifications differ from untimed specifications in that temporal distances between events and states play a role in determining satisfaction. Suppose we want to evaluate a PastMTL formula $\psi_1 \mathbf{S}_{[a:b]} \psi_2$ at the current time point k . Without timing constraints, we need to check the past time points where ψ_2 holds and ensure ψ_2 holds since then. This is insufficient for the timed specification as we also need to check the distance between k and the past time points where ψ_2 holds. The naive way to address this problem is formula discretization, that is, encoding timing constraints as a series of the Previous operator. However, this technique does not scale for large timing constraints, and the monitoring performance is heavily degraded while the timing constraints are getting larger. A better solution for the problem is to keep the history bounded by the window $[k - b, k]$ and check timing constraints retroactively. Although this technique performs better than naive discretization, it still suffers from large timing constraints when the number of events increases in the window. In Section 5.1, we present our performance experiments using existing MTL monitoring tools using these techniques [BHKZ11, BKZ17, BKT17].

In this paper, we propose an alternative technique, called *future temporal marking*, which labels future temporal points with the corresponding information to be used when the future time arrives. For example, consider online monitoring of the PastMTL formula $\varphi := \mathbf{P}_{[a:b]} \psi$. Unlike existing techniques that rely on keeping a bounded history of output values for the formula ψ , our technique proactively marks all future time points in $[k + a, k + b]$ whenever the formula ψ holds at time point k . At a future time point k' , we know the formula φ needs to be evaluated to true if k' is already marked, false otherwise. Such a proactive approach necessitates maintaining a subset of future time points marked for each timed operator as time progresses. This observation is our starting point for constructing a discrete time sequential network that can manipulate sets of integers.

Following the PastLTL construction in Section 3.1, we begin our construction by associating each propositional and untimed subformula with a Boolean state variable and updating them using the same set of update equations in Equation 3.1. Therefore, the construction for the untimed fragment remains unaltered in the discrete setting. For the timed fragment, we create nodes for each timed operator φ that maintains an integer-set (`nset`) valued state variable $\mathbf{V}_k(\varphi) \subseteq [k, \infty)$. This integer set essentially represents a subset of future integer time points, and this property is ensured by intersecting $\mathbf{V}_k(\varphi)$ with $[k, \infty)$ after an update, an operation referred to as *trimming*.

In the following, we explain and formalize how to update timed state variables for each timed operator.

Timed Past Eventually. According to the reflexive PastMTL semantics, the formula $\mathbf{P}_{[a:b]} \psi$ holds at a time point k if the formula ψ holds for some time points in the discrete time interval $[k - b, k - a]$. For the timed Eventually nodes, we want to mark future time points and maintain the integer-set valued state variable $\mathbf{V}(\mathbf{P}_{[a:b]} \psi)$ to update it at

every time point. This behavior is formally captured in the update equation of timed past Eventually nodes as follows:

$$\mathbf{V}_k(\mathbf{P}_{[a:b]} \psi) : \text{nset} \iff \begin{cases} \mathbf{V}_{k-1}(\mathbf{P}_{[a:b]} \psi) : \text{nset} \cup [k+a, k+b] & \text{if } \mathbf{Y}_k(\psi) : \text{bool} \\ \mathbf{V}_{k-1}(\mathbf{P}_{[a:b]} \psi) : \text{nset} & \text{otherwise.} \end{cases} \quad (3.2)$$

where the Boolean-valued function $\mathbf{Y}_k(\psi)$ denotes the output equation of the formula ψ and the initial state \mathbf{V}_0 is defined to be empty. Here, the update equation marks future time points based on the subformula's truth value and timing constraints, while previously marked time points remain marked. It is easy to see that the current time point k must be in $\mathbf{V}_k(\mathbf{P}_{[a:b]} \psi)$ by definition if the formula $\mathbf{P}_{[a:b]} \psi$ holds at k . From this observation, we define the output function $\mathbf{Y}_k(\mathbf{P}_{[a:b]} \psi)$ to be a membership test as follows:

$$\mathbf{Y}_k(\mathbf{P}_{[a:b]} \psi) : \text{bool} \iff k \in \mathbf{V}_k(\mathbf{P}_{[a:b]} \psi) \quad (3.3)$$

Notice that unlike the untimed case, where both state valuations and output values are Boolean, the discrete timed case introduces different data types for valuations and output values. We consider this fact to be the general case for sequential networks and regard the untimed case to be a specialization of the general framework established in this paper.

In the following, we present an example sequential network construction from the **PastMTL** formula $\varphi := \mathbf{P}_{[1:2]} \mathbf{P}_{[1:2]}(p \vee q)$, which has three non-leaf subformulas $\psi_1 : p \vee q$ and $\psi_2 : \mathbf{P}_{[1:2]}(p \vee q)$ including the formula φ itself. Our construction produces one Boolean and two timed nodes in the resulting network, as shown in Table 2 with corresponding update equations and the output function. Notice that the output function $k \in \mathbf{V}_k(\psi_2)$ of $\mathbf{P}_{[1:2]}(p \vee q)$ is embedded into the update equation of $\mathbf{V}_k(\varphi)$ during the construction. Table 3 illustrates an example run over a discrete time behavior from the time index $k = 0$ to 5 over the propositions p and q in the first two rows. The next three rows below denote the valuation of state variables, and the final row denotes the output of the sequential network. Observe that the timed nodes trim their valuations to be a subset of $[k, \infty)$ as time progresses. This is important in practice to keep the size of state valuations small. A minor limitation and our performance experiments related to the size of state valuations are presented in Section 5.1.

Finally note that the formula $\mathbf{P}_{[1:2]} \mathbf{P}_{[1:2]}(p \vee q)$ is semantically equivalent to a simpler formula of $\mathbf{P}_{[2:4]}(p \vee q)$. Therefore, we know it is possible to construct a smaller network to

$\mathbf{V}_k(p) : \text{bool}$	\iff	$\mathbf{X}_k(p) : \text{bool}$
$\mathbf{V}_k(q) : \text{bool}$	\iff	$\mathbf{X}_k(q) : \text{bool}$
$\mathbf{V}_k(\psi_1) : \text{bool}$	\iff	$\mathbf{Y}_k(p) : \text{bool} \vee \mathbf{Y}_k(q) : \text{bool}$
$\mathbf{V}_k(\psi_2) : \text{nset}$	\iff	$\begin{cases} \mathbf{V}_{k-1}(\psi_2) : \text{nset} \cup [k+1, k+2] & \text{if } \mathbf{Y}_k(\psi_1) : \text{bool} \\ \mathbf{V}_{k-1}(\psi_2) : \text{nset} & \text{otherwise.} \end{cases}$
$\mathbf{V}_k(\varphi) : \text{nset}$	\iff	$\begin{cases} \mathbf{V}_{k-1}(\varphi) : \text{nset} \cup [k+1, k+2] & \text{if } \mathbf{Y}_k(\psi_2) : \text{bool} \\ \mathbf{V}_{k-1}(\varphi) : \text{nset} & \text{otherwise.} \end{cases}$
$\mathbf{Y}_k(\varphi) : \text{bool}$	\iff	$k \in \mathbf{V}_k(\varphi) : \text{nset}$

TABLE 2. Sequential network constructed from the formula $\varphi := \mathbf{P}_{[1:2]} \mathbf{P}_{[1:2]}(p \vee q)$ where $\psi_1 := p \vee q$ and $\psi_2 := \mathbf{P}_{[1:2]}(\psi_1)$.

k	—	0	1	2	3	4	5
$\mathbf{X}(p) : \text{bool}$		T	F	F	F	F	F
$\mathbf{X}(q) : \text{bool}$		F	F	F	F	T	F
$\mathbf{V}(p) : \text{bool}$	F	T	F	F	F	F	F
$\mathbf{V}(q) : \text{bool}$	F	F	F	F	F	T	F
$\mathbf{V}(\psi_1) : \text{bool}$	F	T	F	F	F	T	F
$\mathbf{V}(\psi_2) : \text{nset}$	\emptyset	$[1, 2]$	$[1, 2]$	$\{2\}$	\emptyset	$[5, 6]$	$[5, 6]$
$\mathbf{V}(\varphi) : \text{nset}$	\emptyset	\emptyset	$[2, 3]$	$[2, 4]$	$[3, 4]$	$\{4\}$	$[6, 7]$
$\mathbf{Y}(\varphi) : \text{bool}$		F	F	T	T	T	F

TABLE 3. An example execution of the sequential network in Table 2.

monitor this formula. However, in this paper, our constructions follow the formula structure faithfully. Although we do not target the most optimal networks, we still recognize the potential for performance improvements through syntactic optimizations such as formula rewriting and common subformula elimination.

Timed Past Always. According to the reflexive PastMTL semantics, the formula $H_{[a:b]} \psi$ holds at a time point k if the formula ψ holds for all time points in the discrete time interval $[k-b, k-a]$. Similar to the timed past Eventually case, we maintain the integer-set valued state variable $\mathbf{V}(H_{[a:b]} \psi)$ for timed past Always nodes and update it at every time point. To capture this behavior efficiently, we leverage the duality property from Equivalence 2.3(c) and define the update equation for the timed past Always operator as follows:

$$\mathbf{V}_k(H_{[a:b]}) : \text{nset} \Leftarrow \begin{cases} \mathbf{V}_{k-1}(H_{[a:b]}) : \text{nset} \cup [k+a, k+b] & \text{if } \neg \mathbf{Y}_k(\psi) : \text{bool} \\ \mathbf{V}_{k-1}(H_{[a:b]}) : \text{nset} & \text{otherwise.} \end{cases} \quad (3.4)$$

where the Boolean-valued function $\mathbf{Y}_k(\psi)$ denotes the output equation of the formula ψ . But, unlike the timed past Eventually case, the update equation now marks new future time points if the subformula does not hold, and we need to check the absence of current time point k in the $\mathbf{V}_k(H_{[a:b]})$ to output true. The output function is then defined as:

$$\mathbf{Y}_k(H_{[a:b]}) : \text{bool} \Leftarrow k \notin \mathbf{V}_k(H_{[a:b]}) \quad (3.5)$$

The initial state $\mathbf{V}_0(H_{[a:b]})$ is similarly set to the empty set \emptyset . This approach allows the operators $P_{[a:b]}$ and $H_{[a:b]}$ to seamlessly handle vacuous truth situations where no truth

k	—	0	1	2	3	4	5
$\mathbf{X}(p) : \text{bool}$		F	F	T	T	T	F
$\mathbf{V}(H_{[1:2]} p) : \text{nset}$	\emptyset	$[1, 2]$	$[1, 3]$	$[2, 3]$	$\{3\}$	\emptyset	$[6, 7]$
$\mathbf{Y}(H_{[1:2]} p) : \text{bool}$		T	F	F	F	T	T

TABLE 4. An example execution of the sequential network constructed for the formula $\varphi := H_{[1:2]} p$ over an discrete time behavior of the proposition p .

value is assigned before the initial time point. Table 4 illustrates an example run of the sequential network constructed from the formula $\varphi := H_{[1:2]} p$.

Timed Since. According to the reflexive PastMTL semantics, the formula $\psi_1 S_{[a:b]} \psi_2$ holds at a time point k if ψ_2 held at time point k' in the past between time points $k - b$ and $k - a$, and ψ_1 has held continuously from k' to k . The timed Since operation is the most general case for our discrete time PastMTL construction, as the previous cases can be derived from this construction. Intuitively speaking, online monitoring of timed Since nodes using our approach requires marking future time points whenever the formula ψ_2 holds according to timing constraints and removing them when the formula ψ_1 ceases to hold. In other words, timed Since nodes dynamically maintain an integer-set valued state variable based on the output of the formula ψ_2 and timing constraints. These integers represent future time points where the timed Since node might be satisfied, while requiring the subformula ψ_1 to continuously hold until then.

Based on these observations, we formally define the update equation of the sequential network constructed for the formula $\varphi := \psi_1 S_{[a:b]} \psi_2$ as follows:

$$\mathbf{V}_k(\varphi) : \text{nset} \Leftarrow \begin{cases} \mathbf{V}_{k-1}(\varphi) : \text{nset} \cup [k+a, k+b] & \text{if } \mathbf{Y}_k(\psi_1) \wedge \mathbf{Y}_k(\psi_2) \\ [k+a, k+b] & \text{if } \neg \mathbf{Y}_k(\psi_1) \wedge \mathbf{Y}_k(\psi_2) \\ \mathbf{V}_{k-1}(\varphi) : \text{nset} & \text{if } \mathbf{Y}_k(\psi_1) \wedge \neg \mathbf{Y}_k(\psi_2) \\ \emptyset & \text{otherwise.} \end{cases} \quad (3.6)$$

The initial state $\mathbf{V}_0(\psi_1 S_{[a:b]} \psi_2)$ is set to the empty set \emptyset and the output function is similarly defined as a membership test for the timed Since nodes as follows:

$$\mathbf{Y}_k(\psi_1 S_{[a:b]} \psi_2) : \text{bool} \Leftarrow k \in \mathbf{V}_k(\psi_1 S_{[a:b]} \psi_2) \quad (3.7)$$

These definitions complete our discrete time sequential network construction from the reflexive PastMTL specifications. The irreflexive case is similar to the reflexive. The irreflexive update function does not add k to the valuation for the case when $\mathbf{Y}_k(\psi_2)$ holds. Therefore, we can obtain the update function for the irreflexive Since operation from Equation 3.6 by replacing $[k+a, k+b]$ with $(k+a, k+b]$ when $a = 0$ and derive update functions for other irreflexive temporal operators.

Each node in the network synchronously updates its state based on information as defined by their update and output equations. Table 5 illustrates an example run of the sequential network constructed from the formula $\varphi = p S_{[2:3]} q$. At time point 1, the proposition q is true, and we mark future time points $[3, 4]$. These potential satisfaction points for the formula will be kept in the valuation set if the proposition p continues to hold, as realized for time points 3 and 4. The proposition p gets false at time point 5; therefore, potential satisfaction for time points 6 and 7 will not be realized. Note that the valuation set has been trimmed at each time step therefore the valuation set at time point 4 does not contain the time point 3.

Finally, we state the correctness of our construction with the following theorem.

Theorem 3.1. *For any PastMTL formula φ and discrete time behavior w , the sequential network output $\mathbf{Y}_k(\varphi)$ evaluates to true at time point k over w iff $(w, k) \models \varphi$ holds.*

Proof. We only show the correctness of the timed Since case as the propositional and untimed fragment is straightforward and other temporal operators can be derived from timed Since.

k	—	0	1	2	3	4	5
$\mathbf{X}(p) : \text{bool}$		F	F	T	T	T	F
$\mathbf{X}(q) : \text{bool}$		F	T	F	F	T	F
$\mathbf{V}(p \text{ S}_{[2:3]} q) : \text{nset}$	\emptyset	\emptyset	$[3, 4]$	$[3, 4]$	$[3, 4]$	$\{4\} \cup [6, 7]$	\emptyset
$\mathbf{Y}(p \text{ S}_{[2:3]} q) : \text{bool}$		F	F	F	T	T	F

TABLE 5. An example execution of the sequential network constructed for the formula $\varphi := p \text{ S}_{[2:3]} q$ over discrete time behaviors of p and q .

- (\rightarrow) First, we show that $(w, k) \models \psi_1 \text{ S}_{[a:b]} \psi_2$ holds if time point $k \in \mathbf{V}_k(\psi_1 \text{ S}_{[a:b]} \psi_2)$. Assume that $k \in \mathbf{V}_k(\psi_1 \text{ S}_{[a:b]} \psi_2)$. By Definition 3.6 of the update function, this implies that k was added to the valuation set for some $k^\dagger \in [k - b, k - a]$, which satisfies the first and timed condition of the semantic definition in Definition 2.13. Furthermore, this implies ψ_1 holds continuously from the last addition of k until the current time point k as the update function preserves time points from \mathbf{V}_k when ψ_1 holds. Thus, the semantic definition is satisfied in either case, and we conclude $(w, k) \models \psi_1 \text{ S}_{[a:b]} \psi_2$.
- (\leftarrow) Second, we show that $(w, k) \not\models \psi_1 \text{ S}_{[a:b]} \psi_2$ holds if the time point $k \notin \mathbf{V}_k(\psi_1 \text{ S}_{[a:b]} \psi_2)$. Assume that $k \notin \mathbf{V}_k(\psi_1 \text{ S}_{[a:b]} \psi_2)$. There are two possibilities:
 - (i) The time point k is never added to \mathbf{V}_k . Then, there exists no time point $k^\dagger \in [k - b, k - a]$ that satisfies ψ_2 , thus $(w, k) \not\models \psi_1 \text{ S}_{[a:b]} \psi_2$.
 - (ii) The time point k is initially added to \mathbf{V}_k at some time point $k^\dagger \in [k - b, k - a]$ and removed at a later step. This means there exists a time point k^\ddagger such that $k^\dagger < k^\ddagger \leq k$ where ψ_1 does not hold as the update function only removes time points from \mathbf{V}_k when ψ_1 does not hold. Thus, $(w, k) \not\models \psi_1 \text{ S}_{[a:b]} \psi_2$.

The semantic definition is violated in either case, and we conclude $(w, k) \not\models \psi_1 \text{ S}_{[a:b]} \psi_2$. This completes both directions of the equivalence proof, establishing the correctness of the update function for the timed Since operator and our construction. \square

4. DENSE TIME SEQUENTIAL NETWORK CONSTRUCTION

This section explains sequential network constructions over the dense time behaviors from *PastMTL* specifications, extending our discrete time constructions in the previous section.

The first difference between discrete and dense time settings is the representation of temporal behaviors. Discrete time behaviors are naturally described as a finite sequence of atomic observations indexed by discrete time points. Such finite representation is not straightforward for dense time behaviors as we lack a standard successor relation for dense domains. One solution is timed event sequences that sample a finite subset of time points from the dense time domain and order them by the standard ordering relation. However, timed event sequences have a severe limitation in that the state of monitors can be updated only at sample points, and a non-sufficiently sampled time domain may cause the monitor to yield a wrong verdict [Rey16, BKZ18].

An alternative solution is to represent dense time behaviors as finite sequences of time intervals [AH92], that is, non-empty convex subsets of the time domain. Although the traditional continuum representation is more natural and does not have the aforementioned

irregularities of timed event sequences, this representation is much less developed for dense time monitoring algorithms [Rey16, BKZ18]. One possible reason is that any algorithm working on the traditional continuum must deal with all the challenges of real analysis, such as boundary inclusion/exclusion, limit conditions, epsilon neighborhoods, and Zeno paradoxes. However, these mathematical artifacts hardly matter in monitoring and lead to unnecessarily complicated algorithms. Practical monitor implementations thus restrict dense time behaviors to have finite variability—meaning a finite number of discontinuities exists within any bounded interval—and employ a single interval type in practice. These practices are departures from the initial assumptions of the continuum, but nonetheless, they can be formalized using a restriction to finite representation and a coarser topology on the time domain. Yet a deeper discussion is beyond our scope in this paper.

This paper follows the latter solution and represents dense time behaviors as finite sequences of left-open and right-closed intervals. This particular choice of interval type follows from the fact that the *Since* operator is left-continuous [FMNP19], intuitively meaning that the *Since* operation yields and preserves left-open and right-closed interval types on dense domains. Therefore, our approach uniformly treats and requires dense time intervals to be left-open and right-closed for our dense time procedures.

We start our dense time monitor construction by assuming we observe a finite set P of propositions. We consider a dense time Boolean behavior $w : (t_0, t_n] \rightarrow \mathbb{B}^P$ is a function that can be represented as a sequence of constant valued segments such that

$$(t_0, t_1] \rightarrow \mathbf{X}_1; (t_1, t_2] \rightarrow \mathbf{X}_2; \dots; (t_{n-1}, t_n] \rightarrow \mathbf{X}_n$$

where $n \in \mathbb{N}$, $t_0, t_1, \dots, t_n \in \mathbb{Q}$ and $\mathbf{X}_1, \mathbf{X}_2, \dots, \mathbf{X}_n \in \mathbb{B}^P$ are propositional Boolean vectors. For the behavior w , the segmentation $\tau(w)$ is defined to be the set of its endpoints $\{t_0, t_1, t_2, \dots, t_n\}$. We use the notation w_k to refer to the k -th segment of the behavior w , and the function $\text{dom}(k)$ denotes its time domain $(t_{k-1}, t_k]$ for $k = 1, 2, \dots, n$. The duration $|w_k|$ of the segment k is defined to be $t_k - t_{k-1}$. We avoid zero-duration segments in this formulation; thus, time progress is strict.

We then define quantization and condensation as two common operations on dense time behaviors to help analyze the inherent tradeoff between granularity and efficiency in dense time monitoring. Quantization partitions a dense time behavior w into an equivalent and evenly segmented behavior w' such that the duration $|w'_k|$ of each segment k is equal within w' . Quantization requires a fixed segment length δ such that each segment boundary t in the behavior w is an integer multiple of δ . The set $\{k\delta \mid k \in \mathbb{N}\}$ is called the base of the quantization generated by δ . Conversely, condensation merges consecutive segments with the same value, called stuttering segments, into longer segments. This process can be applied iteratively until the behavior is maximally condensed, meaning that containing no stuttering segments. Figure 1 illustrates the application of quantization and condensation to example dense time behaviors.

Dense-time sequential networks process input behavior incrementally, consuming a single constant-valued segment at each step. Crucially, the network operates without requiring prior knowledge of the overall segmentation, nor does it impose constraints on segment boundaries, length, or arrival rate. This flexibility is a key feature of our approach. The network handles varying data streams, making it ideal for real-time processing and systems with asynchronous or irregular updates. While segmentation choices do not affect the network's functionality and output, they may influence monitoring performance considerably. Section 5.3 provides a detailed performance analysis, examining how different segmentation

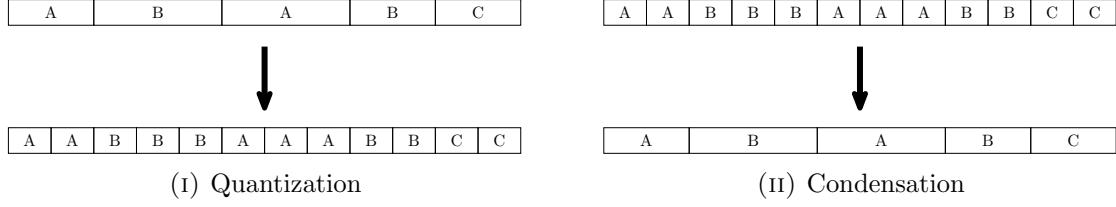


FIGURE 1. Quantization and condensation operations side by side

schemes influence the performance. Yet, ultimately, the user or application side retains control over the segmentation.

In the following section, we explain and formalize how to update sequential network nodes for each operator, extending our discrete time constructions from the previous section.

Dense Time Propositional Fragment. Nodes in our dense time sequential network construction manipulate all current and future time points at a single update step. This is similar to the discrete case except that current time points are no longer isolated. Dense time nodes receive input and produce output for all time points in the current interval.

Dense-time propositional nodes internally represent their state valuation as a subset of the rational numbers, denoted as qset . For a given dense-time segment k , the valuation set $\mathbf{V}_k(p)$ consists of rational time points where the proposition p holds. This extends to Boolean operations naturally, and we define the update equations for both propositional and Boolean nodes as follows:

$$\begin{aligned}
 \mathbf{V}_k(p) : \text{qset} &\iff \{t \in \text{dom}(k) \mid w_p(t) = \text{T}\} \\
 \mathbf{V}_k(\neg\varphi) : \text{qset} &\iff \text{dom}(k) \setminus \mathbf{Y}_k(\varphi) : \text{qset} \\
 \mathbf{V}_k(\varphi_1 \wedge \varphi_2) : \text{qset} &\iff \mathbf{Y}_k(\varphi_1) : \text{qset} \cap \mathbf{Y}_k(\varphi_2) : \text{qset} \\
 \mathbf{V}_k(\varphi_1 \vee \varphi_2) : \text{qset} &\iff \mathbf{Y}_k(\varphi_1) : \text{qset} \cup \mathbf{Y}_k(\varphi_2) : \text{qset}
 \end{aligned} \tag{4.1}$$

where $\text{dom}(k) \subseteq \mathbb{Q}^+$ denotes the domain of the current segment k . For the dense time setting, the output of each segment is a subset of its time domain, where membership indicates the formula's validity at that particular point in time. The output function for propositional nodes is then trivially defined as follows:

$$\mathbf{Y}_k(\varphi) : \text{qset} \iff \mathbf{V}_k(\varphi) : \text{qset} \tag{4.2}$$

Therefore, the network efficiently computes and propagates truth values of dense time points within a segment in a single, unified update.

Dense Timed Since. We handle dense timed Since nodes similar to the discrete time construction with a few modifications for the dense time setting.

First, the network node for timed Since needs to process the output values of their operand nodes over constant segments. Yet, generally, constant segments in the operand outputs are not synchronized inside the current segment. Therefore, we must synchronize dense time output valuations from operands to provide constant-valued local segments before feeding them into our node update procedure. To this end, we introduce a secondary local index l that indicates the position of a local segment in the current global segment. For example, consider that we are processing the k -th segment that consists of L_k constant-valued local segments over the operand values ψ_1 and ψ_2 for the formula $\psi_1 \mathbf{S}_{[a:b]} \psi_2$. The notation

$\mathbf{V}_{k,l}$ denotes the valuation of l -th constant local segment for both operands in the k -th global segment for $l = 1 \dots L_k$, and similarly $\mathbf{Y}_{k,l}$ denotes its output.

The example in Table 6 illustrates the synchronization operation over dense time behaviors of sub-formulas ψ_1 and ψ_2 . Dense time behaviors of the operands are sequentially given as four global segments in the example, and we locally synchronize them before local updates of the timed Since operation. Synchronization may lead to further segmentation on the timeline in which both behaviors hold constant, as shown in the row denoted by $\psi_1 \parallel \psi_2$. In the worst case, this leads to $|\mathbf{Y}_k(\psi_1)| + |\mathbf{Y}_k(\psi_2)| - 1$ local segments inside the global segment k . For this example, the first three global segments are divided into four local segments, and the last is divided into two local segments.

Second, node updates within each global segment are performed through local update steps. Similar to the discrete time update function in Equation 3.6, the update function for dense timed Since nodes $\varphi = \psi_1 \mathbf{S}_{[a:b]} \psi_2$ for the local segment (k, l) is defined as follows:

$$\mathbf{V}_{k,l}(\varphi) : \text{qset} \Leftarrow \begin{cases} \mathbf{V}_{k, l-1}(\varphi) : \text{qset} \cup (t + a, t' + b] & \text{if } y_{k,l}(\psi_1) \wedge y_{k,l}(\psi_2) \\ (t' + a, t' + b] & \text{if } \neg y_{k,l}(\psi_1) \wedge y_{k,l}(\psi_2) \\ \mathbf{V}_{k, l-1}(\varphi) : \text{qset} & \text{if } y_{k,l}(\psi_1) \wedge \neg y_{k,l}(\psi_2) \\ \emptyset & \text{otherwise.} \end{cases} \quad (4.3)$$

where $\text{dom}(k, l) = (t, t']$ and $y_{k,l}$ is the Boolean value for the constant-valued local segment (k, l) for the operands. Local updates follow a strict sequential order like global update steps, with the final local step serving as the initial step for the subsequent segment such that $\mathbf{V}_{k,L_k} = \mathbf{V}_{k+1,0}$. The irreflexive Since operation is left-continuous by the semantic definition in Equation 2.10, meaning that, for all behaviors w and time points t ; $(w, t) \models \varphi$ implies there exists $t' < t$ such that for all $t'' \in (t', t)$. $(w, t) \models \varphi$. It is easy to see that the local update function preserves the left-continuity of the valuation set.

Finally, we combine the outputs of local updates to prevent timeline fragmentation at the network interface, which may be problematic when the network's output is used to feed another network in a compositional way. Analogously to the discrete time output function defined in Equation 3.7, the output function for a local segment is defined using the set intersection operation as follows:

$$\mathbf{Y}_{k,l}(\varphi) : \text{qset} \Leftarrow \mathbf{V}_{k,l}(\varphi) : \text{qset} \cap \text{dom}(k, l) : \text{qset} \quad (4.4)$$

This set intersection operation symbolically checks membership for all points within the current time segment and yields all time instants that satisfy the formula for the segment. The final computation step combines the outputs of individual local steps within each global segment and yields the output of the global segment as a whole. This step is achieved through the union operation, effectively preventing any timeline fragmentation that could otherwise arise from the local synchronization approach. Leveraging these insights, we formally define the output of the global segment k as a union of the outputs of all L_k local segments as follows:

$$\mathbf{Y}_k(\varphi) : \text{qset} \Leftarrow \mathbf{Y}_{k,1}(\varphi) : \text{qset} \cup \mathbf{Y}_{k,2}(\varphi) : \text{qset} \cup \dots \cup \mathbf{Y}_{k,L_k}(\varphi) : \text{qset} \quad (4.5)$$

Therefore, a dense time node receives the inputs for the current time interval and yields an output for the same interval as the sequential model of computation dictates.

Table 6 illustrates how our sequential network processes a dense time behavior for the formula $\varphi = \psi_1 \mathbf{S}_{[18:24]} \psi_2$ where ψ_1 and ψ_2 represent subformulas over dense time. Each row corresponds to a step in the processing, progressing through global segments defined

Step(k) :	1	2	3	4
Domain :	(0, 30]	(30, 47]	(47, 75]	(75, 99]
ψ_1 :	$(0, 7] \rightarrow F$; $(7, 30] \rightarrow T$;	$(30, 35] \rightarrow T$; $(35, 39] \rightarrow F$; $(39, 47] \rightarrow T$;	$(47, 49] \rightarrow T$; $(49, 63] \rightarrow F$; $(63, 75] \rightarrow T$;	$(75, 99] \rightarrow T$;
ψ_2 :	$(0, 3] \rightarrow F$; $(3, 8] \rightarrow T$; $(8, 30] \rightarrow F$;	$(30, 38] \rightarrow F$; $(38, 39] \rightarrow T$; $(39, 47] \rightarrow F$;	$(47, 70] \rightarrow F$; $(70, 75] \rightarrow T$;	$(75, 89] \rightarrow T$; $(89, 99] \rightarrow F$;
$\psi_1 \parallel \psi_2$:	$y_{\psi_1} y_{\psi_2}$ (0, 3] : F F (3, 7] : F T (7, 8] : T T (7, 30] : T F	$y_{\psi_1} y_{\psi_2}$ (30, 35] : T F (35, 38] : F F (38, 39] : F T (39, 47] : T F	$y_{\psi_1} y_{\psi_2}$ (47, 49] : T F (49, 63] : F F (63, 70] : T F (70, 75] : T T	$y_{\psi_1} y_{\psi_2}$ (75, 89] : T T (89, 99] : T F
$\mathbf{V}_{k,l}(\varphi)$:	1 : \emptyset 2 : $\{(25, 31]\}$ 3 : $\{(25, 32]\}$ 4 : $\{(25, 32]\}$	1 : $\{(30, 32]\}$ 2 : \emptyset 3 : $\{(57, 63]\}$ 4 : $\{(57, 63]\}$	1 : \emptyset 2 : \emptyset 3 : \emptyset 4 : $\{(88, 99]\}$	1 : $\{(88, 113]\}$ 2 : $\{(89, 113]\}$
$\mathbf{Y}_k(\varphi)$:	$\{(25, 30]\}$	$\{(30, 32]\}$	\emptyset	$\{(88, 99]\}$

TABLE 6. An example run of the network constructed for the formula $\varphi = \psi_1 \mathbf{S}_{[18:24]} \psi_2$ over dense time behaviors of ψ_1 and ψ_2 .

by the input behavior and the timing window of the Since operator. Within each global segment, the table breaks down the output $\mathbf{Y}(\varphi)$ into local valuations (\parallel) for individual time periods. These local valuations depend on the satisfaction of both subformulas $\mathbf{Y}(\psi_1)$ and $\mathbf{Y}(\psi_2)$ within the corresponding period and the timing constraints of the Since operation. The final output $\mathbf{Y}(\varphi)$ for each global segment is obtained by joining the local outputs, reflecting the overall periods where the formula φ holds based on the combined behavior of ψ_1 and ψ_2 across the global segment.

Finally, we establish the correctness of our dense time construction. Our strategy involves two steps. First, we reduce the dense-time behavior to an equivalent, evenly-segmented form compatible with the timing constraints specified in the formula. Then, we prove the correctness of our procedure over this reduced behavior, without loss of generality. The structure of the dense-time proof deliberately is similar to the discrete case.

Proposition 4.1. *Given a PastMTL formula φ , there exists a base of quantization for any dense time behavior w is closed under dense time sequential network update functions.*

Proof. Let $\delta \in \mathbb{Q}^+$ be the greatest rational common divisor of all segment boundaries in the behavior w and timing constraints in the formula φ . The base of quantization generated by δ is trivially closed under addition; thus dense time Since update function in Equation 4.3 does not generate new segment boundaries outside the existing base. Since Boolean update functions trivially preserve the existing base, the proof is complete. \square

Theorem 4.2. *For any PastMTL formula φ and dense time behavior w , the sequential network output $\mathbf{Y}_k(\varphi)$ evaluates to true at time point k over w iff $(w, k) \models \varphi$ holds.*

Proof. Let an evenly segmented behavior w' that is equivalent to the behavior w and compatible with the formula φ by Proposition 4.1. We assume the fixed segment length $\delta = 1$ without loss of generality and proceed similarly to the proof of discrete time Theorem 3.1. Boolean operations are straightforward. The case for dense time Since is as follows:

(\rightarrow) First, we show that $(w, t) \models \psi_1 \mathbf{S}_{[a:b]} \psi_2$ holds for all $t \in (k, k+1]$ if a segment $(k, k+1] \subseteq \mathbf{V}_k(\psi_1 \mathbf{S}_{[a:b]} \psi_2)$. Assume that $(k, k+1] \subseteq \mathbf{V}_k(\psi_1 \mathbf{S}_{[a:b]} \psi_2)$. By Definition 2.10 of the dense time update function, at least one of the following cases holds:

- (i) The segment $(k, k+1]$ was added to the valuation set by a segment $(k^\dagger, k^\dagger + 1] \subseteq (k-b, k-a]$ where φ_2 and $\neg\varphi_1$ holds. Furthermore, the formula φ_1 holds for all points in $(k^\dagger + 1, k+1]$. Then, the intersection of $(k-b, k-a]$ and $[t-b, t-a]$ for all $t \in (k, k+1]$ is non-empty. Hence, $(w, t) \models \psi_1$ for all $t \in (k, k+1]$.
- (ii) The segment $(k, k+1]$ was added to the valuation set by a segment $(k^\dagger, k^\dagger + 1] \subseteq (k-b, k-a+1]$ where φ_2 and φ_1 holds. Furthermore, the formula φ_1 holds for all points in $(k^\dagger, k+1]$. Then, the intersection of $(k-b, k-a+1]$ and $[t-b, t-a]$ for all $t \in (k, k+1]$ is non-empty. Hence, $(w, t) \models \psi_1$ for all $t \in (k, k+1]$.

The semantic definition of the Since operation is satisfied in either case, and we conclude $(w, t) \models \psi_1$ for all $t \in (k, k+1]$.

(\leftarrow) Second, we show that $(w, t) \not\models \psi_1 \mathbf{S}_{[a:b]} \psi_2$ holds for all $t \in (k, k+1]$ if a segment $(k, k+1] \not\subseteq \mathbf{V}_k(\psi_1 \mathbf{S}_{[a:b]} \psi_2)$. Assume that $(k, k+1] \not\subseteq \mathbf{V}_k(\psi_1 \mathbf{S}_{[a:b]} \psi_2)$. There are two possibilities:

- (i) The segment $(k, k+1]$ is never added to \mathbf{V}_k . Then, there exists no segment that satisfies ψ_2 and timing constraints, thus $(w, k) \not\models \psi_1 \mathbf{S}_{[a:b]} \psi_2$.
- (ii) The segment $(k, k+1]$ is initially added to \mathbf{V}_k at some segment k^\dagger and removed at a later step. This means there exists a segment k^\ddagger between $k^\dagger < k^\ddagger \leq k$ where ψ_1 does not hold as the update function only removes time points from \mathbf{V}_k when ψ_1 does not hold. Thus, $(w, k) \not\models \psi_1 \mathbf{S}_{[a:b]} \psi_2$.

The semantic definition is violated in either case, and we conclude $(w, k) \not\models \psi_1 \mathbf{S}_{[a:b]} \psi_2$.

This completes both directions of the equivalence proof, establishing the correctness of the dense time update function for the timed Since operator and our construction. \square

The quantization technique employed in the proof is not practical, however. In practice, the number of quantized segments may be very large and sensitive to the specific numeric constants present in the behavior and timing constraints. This would bring a higher and unnecessary computational burden in many dense time monitoring applications. Therefore, our dense time procedure performs a lazy and local synchronization on the fly, only dividing time intervals as needed during the evaluation process.

5. IMPLEMENTATION AND EVALUATION

This section presents a comparative performance and scalability analysis of the Reelay monitoring library that implements sequential network-based monitors as explained in this paper. Reelay² provides a comprehensive solution for specification-based monitoring of temporal behaviors, offering flexibility and extensibility at its core. Its latest version is

²<https://github.com/doganulus/reelay>

written as a header-only C++ template library. Reelay implements sequential networks as an acyclic computation graph using runtime polymorphism in C++, which provides a flexible structure and easier usage but incurs virtual function indirection overhead. The software design permits users to customize the types for input streams, allowing them to opt for their preferred representation formats for temporal behaviors, provided they create the necessary adapter code. Furthermore, as a standalone monitoring tool, Reelay releases include a C++ application `ryjson`, which leverages the `reelay` library to monitor the newline-delimited sequences of JSON documents, which represents discrete and dense time behaviors. This application is exclusively utilized for all performance experiments conducted in this study.

We start our analysis with some basic experiments to demonstrate the benefits and limitations of the MTL monitoring technique explained in this paper. In particular, we compare existing techniques implemented in monitoring tools against certain challenging properties on synthetic and adversarial inputs. This analysis clarifies the importance and effectiveness of our approach in tackling large timing constraint issues, especially when compared to contemporary state-of-the-art solutions. In this part, we also compare discrete and dense time monitoring, delving into their strengths and weaknesses within various monitoring contexts based on performance results.

The second and third parts of the analysis demonstrate the typical performance of discrete and dense time monitors over Timescales benchmarks [Ulu19]. Discrete time benchmarks compare Reelay with other publicly available MTL monitoring tools, whereas dense time benchmarks are compared with the discrete case. In all our experiments, we report performance results from our Linux-based containerized benchmarking environment, which runs on a six-core 3.80GHz Intel Xeon W-2235 CPU. Benchmark scripts and repeatability instructions are available in our code repository³.

5.1. Basic Experiments. As detailed in Sections 3 and 4, our monitor construction technique generates a set of nodes with update and output functions based on the syntax tree of the `PastMTL` formula. These nodes form an acyclic computation graph where each performs certain Boolean and interval set operations depending on the node type and the time model at every computation step. This section compares our approach with other alternative approaches implemented in two publicly available monitoring tools, MonPoly [BHKZ11, BKZ17] and Aerial [BKT17], that support online `PastMTL` monitoring. Both tools use a sample-based time model, equivalent to the discrete model if there is a sample for each discrete time point. Conversely, their sample-based model and our interval-based model for dense time behaviors are inherently different. Therefore, we restrict our evaluation to discrete time behaviors only for a fair comparison.

To systematically evaluate the scalability of timed monitoring approaches under different temporal constraints and behavior lengths, we conduct a number of experiments using three parameterized `PastMTL` properties. We define three variants for each property, each encompassing a tenfold increase in timing constraints, spanning from ten to thousand time units. These properties are tested over temporal behaviors whose lengths range from ten thousand to one million. Ideally, we expect the performance of monitoring tools to scale linearly to the behavior length while exhibiting constant execution times with respect to varying timing constraints—a particularly important metric for real-time systems where specified timing constraints can be significantly larger than the system’s base time unit.

³<https://github.com/doganulus/timescales>

Table 7 presents our evaluation results for these performance experiments. First, the property $\text{QPR} : H((r \wedge \neg q \wedge P q) \rightarrow (p \text{ S}_{[1..u]} q))$ is a typical *PastMTL* property where we consider three variants, QPR10 , QPR100 , and QPR1000 , whose timing parameters are defined as $[3:6]$, $[30:60]$, and $[300:600]$, respectively. We see that *MonPoly* and *Reelay* handle longer temporal behaviors and large timing constraints as expected for QPR property. *Aerial* also handles longer behaviors as expected. Yet it does not scale for large timing constraints as it uses naive discretization for handling timing constraints. For these properties and behaviors, *Reelay* appears 10-15 times faster than *MonPoly*, yet the performance difference is still in the margin of the implementation details between tools—especially regarding the programming language choices of C++ (*Reelay*) and OCaml (*MonPoly*, *Aerial*).

Second, the property $\text{PandQ} : p \text{ S}_{[a..b]} q$ is a basic property where we consider three variants, PandQ10 , PandQ100 , and PandQ1000 , whose timing parameters are defined as $[1:6]$, $[1:60]$, and $[1:600]$, respectively. We evaluate these three variants over a specific temporal behavior where the proposition q occurs frequently. This combination attacks *MonPoly*'s approach of keeping bounded history windows for timed nodes. The performance results in Table 7 show that *MonPoly* does not scale for this particular case as *MonPoly* explicitly stores all occurrences of the second argument of *Since* operation in the history window bounded by the constraint b . For *Reelay*, frequent occurrence of q leads to overlapping marked periods in the future that we can merge, thus keeping the size of the valuation set small. Therefore, our approach performs well in this case and requires a constant time as the timing constraint b gets larger.

However, for the third property, we attack our own method by a very specific adversarial case that involves (1) a formula $P_{[a..b]} q$ with a very precise and large timing constraint such that $b - a \ll b$, and (2) an input behavior of the proposition q holds very frequently such as on every other time point. This pathological scenario is captured by the property Delay where we consider three variants, Delay10 , Delay100 , and Delay1000 , whose timing parameters are defined as $[6:6]$, $[60:60]$, and $[600:600]$, respectively. The performance results in Table 7 show that the Delay property does not scale for all three tools. For *Reelay*, this case leads to linear growth of the valuation set with respect to the constraint b , thus

	Aerial			MonPoly			Reelay		
	10K	100K	1000K	10K	100K	1000K	10K	100K	1000K
QPR10	0.031	0.303	3.413	0.025	0.285	3.062	0.020	0.037	0.255
QPR100	0.270	2.658	27.418	0.024	0.274	2.861	0.020	0.036	0.263
QPR1000	3.859	38.345	315.566	0.026	0.275	2.911	0.020	0.036	0.260
PandQ10	0.038	0.267	2.544	0.032	0.349	3.535	0.021	0.046	0.367
PandQ100	0.243	2.275	22.723	0.066	0.639	6.500	0.021	0.055	0.371
PandQ1000	3.215	32.039	401.784	0.339	3.664	36.786	0.021	0.048	0.365
Delay10	0.039	0.252	2.538	0.028	0.300	3.048	0.021	0.046	0.387
Delay100	0.215	2.126	21.015	0.036	0.404	4.243	0.026	0.133	1.280
Delay1000	2.952	29.339	294.031	0.142	1.551	15.304	0.087	1.100	10.523

TABLE 7. Total execution times in seconds for discrete time monitoring tools over three different properties and behavior lengths.

deteriorating the performance as b grows larger. We present this case as a pathological example of timed monitoring applications.

5.2. Discrete Time Benchmarks. This section presents a comparative analysis of performance benchmarks for our discrete time monitors against existing publicly available tools. To provide a rigorous evaluation, we leverage the Timescales benchmark generator [Ulu19], enabling the generation of semi-randomized temporal behaviors satisfying predefined temporal logic formulas at every time point. Table 8 details the 10 Timescales properties used in our experiments, representing real-world scenarios and as reliable proxies for assessing typical monitoring tool performance. Following the methodology from the previous section, we evaluate each property through three variants featuring a tenfold difference in their timing parameters. These benchmarks demonstrate the performance and scalability of monitoring tools over typical scenarios, complementing our findings in Section 5.1.

Table 9 presents a performance comparison of the monitoring tools (Aerial, MonPoly, Reelay) across Timescales properties over discrete time behaviors with a length of 1 million. These results reinforce the evidence of the practical scalability of Reelay and MonPoly in the face of varying timing constraints, as both maintain near-constant execution times over many different properties. Conversely, Aerial demonstrates its limitation in handling large constraints and does not scale at all. Regarding absolute speed, Reelay’s performance is consistently faster than MonPoly in these benchmarks. As the speed of executions is very important for many online and offline applications, we must ensure monitoring tools are optimized for speed. Therefore, we strongly advocate for compiled and system languages in developing monitoring tools, ensuring they can effectively meet the stringent performance demands of modern applications.

5.3. Dense Time Benchmarks. In this section, we present a comprehensive performance analysis and benchmarking of our dense time monitor implementations. We use the same set of Timescales properties and discrete time behaviors defined in Section 5.2 for dense time experiments after *condensing* these behaviors, to enable a direct comparison of time models and their trade-offs. Recall that we interpret the discrete time point k with a value v as a constant segment of $(k, k + 1]$ with the value v under the dense time setting. To enrich our experiments, we consider an additional parameter that controls the maximum duration for condensed periods. This parameter is particularly helpful for simulating real-time systems that must respond at a minimum rate. In the extreme case, setting the parameter to 1 disables condensation, resulting in the original discrete time representation. Conversely, setting it to a sufficiently large value eliminates all stuttering periods, leading to the most condensed representation. It is easy to see that condensation is a lossless compression operation for dense time behaviors.

Table 10 presents the performance of our dense time monitors on dense time Timescales benchmarks. These benchmarks were generated using different values for the condensation parameter, as explained earlier. The benchmark sets, named **Dense1**, **Dense10**, and **Dense100**, represent scenarios where the maximum duration of a condensed segment is restricted to 1, 10, and 100 time units, respectively. The **Dense** benchmark set represents the scenario with no stuttering periods in input behaviors. Len columns in Table 10 indicate the length of the condensed behavior, which originally had a length of 1 million units. Hence, condensation achieves a compression ratio between 56% and 99% over this particular set of

Timescales Properties	MTL Formulas	
Absence After Q Always the case that the proposition p does not occur at least for b time units after the proposition q occurs.	$H(P_{[:b]} q \longrightarrow (\neg p S q))$	(AbsentAQ)
Absence Before R Always the case that the proposition p does not occur at least for b time units before the proposition r occurs.	$H(r \longrightarrow H_{[:b]} \neg p)$	(AbsentBR)
Absence Between Q and R Always the case that the proposition p does not occur between propositions q and r and the duration between q and r is in a and b time units.	$H((r \wedge \neg q \wedge P q) \longrightarrow (\neg p S_{[a:b]} q))$	(AbsentBQR)
Universality After Q Always the case that the proposition p always occurs at least for b time units after the proposition q occurs	$H(P_{[:b]} q \longrightarrow (p S q))$	(AlwaysAQ)
Universality Before R Always the case that the proposition p always occurs at least for b time units before the proposition r occurs.	$H(r \longrightarrow H_{[:b]} p)$	(AlwaysBR)
Universality Between Q and R Always the case that the proposition p always occurs between propositions q and r , and the duration between q and r is in a and b time units.	$H((r \wedge \neg q \wedge P q) \longrightarrow (p S_{[a:b]} q))$	(AlwaysBQR)
Recurrence Globally Always the case that the proposition p occurs at least for every b time unit.	$H(P_{[:b]} p)$	(RecurGLB)
Recurrence Between Q and R Always the case that the proposition p occurs at least for every b time unit between propositions q and r .	$H((r \wedge \neg q \wedge P q) \longrightarrow (P_{[:b]}(p \vee q) S q))$	(RecurBQR)
Response Globally Always the case that the proposition s responds to the proposition p in a and b time units.	$H((s \longrightarrow P_{[a:b]} p) \wedge \neg(\neg s S_{[b:]} p))$	(RespondGLB)
Response Between Q and R. Always the case that the proposition s responds to the proposition p in a and b time units between propositions q and r .	$H((r \wedge \neg q \wedge P q) \longrightarrow ((s \longrightarrow P_{[a:b]} p) \wedge \neg(\neg s S_{[b:]} p)))$	(RespondBQR)

TABLE 8. Parameterized Timescales Properties [Ulu19]

	Aerial	MonPoly	Reelay		Aerial	MonPoly	Reelay
AbsentAQ10	3.277	6.179	0.396	AlwaysAQ10	3.347	6.241	0.381
AbsentAQ100	15.378	6.141	0.369	AlwaysAQ100	15.210	6.234	0.350
AbsentAQ1000	166.553	6.200	0.366	AlwaysAQ1000	166.587	6.188	0.349
AbsentBQR10	3.768	7.513	0.529	AlwaysBQR10	3.332	5.910	0.408
AbsentBQR100	17.987	7.544	0.487	AlwaysBQR100	17.702	5.968	0.417
AbsentBQR1000	191.937	7.327	0.484	AlwaysBQR1000	193.257	5.998	0.412
AbsentBR10	2.979	5.851	0.422	AlwaysBQR10	3.673	7.659	0.511
AbsentBR100	14.901	5.761	0.420	AlwaysBQR100	15.479	7.688	0.473
AbsentBR1000	166.440	5.784	0.421	AlwaysBQR1000	166.743	7.646	0.466
RecurGLB10	2.867	4.958	0.304	RecurBQR10	4.274	7.890	0.626
RecurGLB100	14.525	4.790	0.233	RecurBQR100	20.961	7.769	0.565
RecurGLB1000	164.242	4.826	0.227	RecurBQR1000	220.845	7.753	0.549
RespondGLB10	4.968	6.368	0.566	RespondBQR10	5.709	9.525	0.844
RespondGLB100	31.062	6.255	0.463	RespondBQR100	31.797	9.586	0.749
RespondGLB1000	356.832	6.178	0.450	RespondBQR1000	363.070	9.481	0.748

TABLE 9. Total execution times in seconds for discrete time monitoring tools over Timescales benchmarks

behaviors. Note that this is an important practical advantage for the dense time setting in reducing log storage requirements significantly.

First, we present execution times for our dense time monitors for each property using uncondensed (discrete) input behaviors under Column **Dense1**. For comparison, the discrete time performance for the same inputs is shown under Column **Discrete**. As expected, the discrete time implementation is 3-5 times faster than the dense time implementation due to the absence of synchronization and its use of more primitive data types and fewer memory accesses. Then, we observe that dense time performance improves as we increase the level of condensation through **Dense10**, **Dense100**, and **Dense** columns. This is because our dense time monitor can process condensed behaviors directly and such symbolic treatment corresponds to processing multiple unit intervals at one step. Consequently, dense time monitors are better suited for high-frequency temporal behaviors with smaller time steps with fewer changes. Therefore, dense time performance surpasses discrete time performance for highly condensed behaviors, demonstrating the effectiveness of our approach.

6. RELATED WORK

A pioneering work in temporal logic monitoring is the work by Moszkowski to specify and interpret properties of digital circuits and other discrete systems in [Mos84]. Following his separation theorem for temporal logic, Gabbay differentiated the role of past and future temporal operators for monitoring applications in [Gab89]. These early works and separating the role of past and future temporal logic would be critical for developing efficient algorithms for temporal logic monitoring as Havelund and Roşu introduced a simple and

effective dynamic programming technique to construct online monitors directly from PastLTL specifications in [HR04].

The previously mentioned works, along with others like [MS03, EFH⁺03, HJL03, FK09, BLS11, HPU17], laid the groundwork for temporal logic monitoring, but they primarily focused on discrete untimed models. Real-time systems necessitate precise timing constraints that discrete models struggle to represent accurately. Naive discretization techniques exist, but they introduce significant computational overhead. Additionally, improper discretization can lead to a loss of crucial temporal information, hindering the ability to monitor real-time systems effectively. Following their successful applications in model checking and formal

	Discrete	Dense1	Dense10		Dense100		Dense	
	Time	Time	Len	Time	Len	Time	Len	Time
AbsentAQ10	0.168	0.767	345K	0.276	333K	0.269	333K	0.269
AbsentAQ100	0.157	0.731	303K	0.219	259K	0.185	258K	0.183
AbsentAQ1000	0.154	0.731	296K	0.211	255K	0.178	250K	0.176
AbsentBQR10	0.264	0.743	444K	0.386	444K	0.386	444K	0.385
AbsentBQR100	0.242	0.668	137K	0.120	59K	0.069	59K	0.068
AbsentBQR1000	0.239	0.657	95K	0.081	15K	0.029	6K	0.023
AbsentBR10	0.181	0.607	345K	0.243	333K	0.236	333K	0.236
AbsentBR100	0.180	0.604	303K	0.210	260K	0.186	258K	0.184
AbsentBR1000	0.181	0.603	296K	0.206	254K	0.183	251K	0.181
AlwaysAQ10	0.168	0.804	345K	0.279	333K	0.268	333K	0.267
AlwaysAQ100	0.159	0.761	303K	0.215	260K	0.174	258K	0.173
AlwaysAQ1000	0.156	0.758	295K	0.205	255K	0.167	250K	0.163
AlwaysBQR10	0.248	1.033	444K	0.514	444K	0.512	444K	0.512
AlwaysBQR100	0.232	0.949	137K	0.159	59K	0.087	59K	0.087
AlwaysBQR1000	0.230	0.943	95K	0.109	15K	0.034	6K	0.025
AlwaysBR10	0.180	0.642	345K	0.239	333K	0.231	332K	0.232
AlwaysBR100	0.180	0.636	303K	0.204	260K	0.176	258K	0.175
AlwaysBR1000	0.179	0.634	296K	0.197	255K	0.172	251K	0.169
RecurGLB10	0.137	0.404	327K	0.166	327K	0.167	327K	0.166
RecurGLB100	0.106	0.362	117K	0.063	39K	0.037	39K	0.036
RecurGLB1000	0.104	0.359	93K	0.051	12K	0.023	6K	0.020
RecurBQR10	0.303	1.067	336K	0.426	336K	0.421	335K	0.421
RecurBQR100	0.278	0.993	117K	0.142	39K	0.066	39K	0.65
RecurBQR1000	0.276	0.991	93K	0.111	12K	0.032	4K	0.023
RespondGLB10	0.262	0.974	375K	0.454	374K	0.451	374K	0.452
RespondGLB100	0.220	0.866	124K	0.139	45K	0.072	44K	0.072
RespondGLB1000	0.217	0.863	94K	0.100	13K	0.032	4K	0.023
RespondBQR10	0.414	1.591	487K	0.863	487K	0.862	487K	0.859
RespondBQR100	0.379	1.492	147K	0.253	69K	0.142	69K	0.141
RespondBQR1000	0.383	1.485	97K	0.161	16K	0.045	7K	0.031

TABLE 10. Total execution times in seconds for discrete and dense time monitoring over Timescales benchmarks using Reelay monitoring tool

verification, real-time logics [Koy90, CHR91, AH94] have been employed for specification-based monitoring over two different temporal representations [BKZ18]. The first approach models continuous/dense time behaviors as sequences of isolated events (samples) with timestamps and applies to monitor them [TR05, Dru06, HOW14, BKMZ15]. While this sample based approach yields computationally tractable monitoring algorithms due to its reliance on discrete time representations, its effectiveness relies on the initial discretization, which may reside outside the control of the monitoring framework. This is a potential source of inaccuracies and inefficiencies. In contrast, the second branch focuses on timed state sequences [MNP05, MNP06, CPS09b, BN13], which are more natural to represent physical phenomena. Monitoring algorithms for this representation exist on a spectrum of trade-offs between practicality and mathematical rigor. Our time model in this paper is compatible with the previous works of [ACM97, UFAM14, UFAM16, UM18], which offers a balanced position on this spectrum without compromising mathematical rigor and practicality.

The monitor construction technique presented in [HR04] has been proved very fruitful and extended for timed and quantitative properties in several subsequent works [RFB14, DHF14, BKMZ15, CM20, HPU17, HKO⁺23, MW20, MCW23] and implemented in several monitoring tools [CPS09a, BHKZ11, BKT17, HPU18]. Other approaches for online MTL monitoring include automata based [MNP05, BBKT05, HOW14], tester based [MNP06], and the incremental marking procedures [MN13]. These approaches rely on a point-based continuous-time model compared to our period (segment) based dense time model, as discussed in Section 4. This is a fundamental theoretical difference, and a comprehensive comparison of point-based and period-based time models can be found in [vB91]. Moreover, no implementation is publicly available or maintained for these automata and tester-based monitors to allow practical comparisons. The implementation of the incremental algorithm has been reported to work only with an external simulator in [MN13] and is not available in the latest version [NLM⁺18]. Therefore, implementability remains a significant concern, whereas our network-based technique offer a more practical and easily-implementable solution for dense time monitoring. Finally, it is worth mentioning that the recent works also consider the monitor verifiability and explainability [SBKT19, CM20, BDH⁺22, LHR⁺23]. The explicit state representation and ease of inspection in sequential network-based monitors could contribute to this line of research.

Synchronous dataflow programming languages, such as Lustre [HCRP91] and Esterel [BG92], and stream runtime verification frameworks [dSS⁺05, PGMN10, CHL⁺18, GS18, FFS⁺19] offer alternative monitoring solutions. These solutions leverage procedural domain-specific languages (DSLs) to describe temporal properties, contrasting with the declarative logic formulas used in temporal logic monitoring. While procedural languages hold the advantage of familiarity for many software engineers, existing safety standards often advocate for declarative formal semantics due to their unambiguous nature and verifiability. This motivates the common practice of embedding temporal logic into procedural monitoring frameworks, with various levels of timing constraints support. However, existing stream real-time runtime verification solutions are still limited by the aforementioned complications of time-event sequences. This paper shows that time-state sequences can play nicely with the synchronous paradigm under careful design decisions; hence, it is not an inherent limitation, and our procedures can easily be embedded into these frameworks.

7. CONCLUSION

The ability to monitor diverse data streams, ranging from high-frequency, evenly sampled data to low-frequency, event-driven data, is equally critical to understanding and analyzing complex engineered systems. Discrete and dense time models effectively capture these distinct data characteristics. Discrete time models excel at handling high-frequency, evenly sampled data, where data points are collected at fixed intervals. Conversely, dense time models are better suited for unevenly sampled, variable-frequency data, enabling the capture of physical phenomena and long-term relationships. To offer a comprehensive solution, monitoring tools must seamlessly integrate both discrete and dense time models.

This paper presented a temporal logic monitoring solution to handle both time models in a unified manner. By leveraging the sequential model of computation, we constructed sequential networks from past metric temporal properties. Our key technique, future temporal marking, employs a symbolic representation of timelines using interval structures, enabling efficient analysis and implementation. Unlike existing approaches, our technique effectively handles the complexities of dense time models by discretizing time into variable-duration symbolic steps. This approach clearly delineates our assumptions and distinctions from other methods. We comprehensively evaluated our methods through extensive benchmarking and comparative analysis across various properties and scenarios.

Future work will explore extending our framework to encompass diverse temporal logic variants, including first-order, robust, and probabilistic extensions. This entails generalizing our Boolean algebra-based approach by defining suitable update and output functions for other compatible algebraic structures, along with efficient supporting data structures. We will continue to make these techniques and libraries openly available to facilitate adoption and community contributions. Our long term goal is a unified monitoring framework capable of handling a broad spectrum of temporal logic flavors, thereby addressing the fragmentation prevalent in current implementations.

In this paper, we primarily focused on constructing monitors for individual temporal properties. However, many real-world applications require the simultaneous monitoring of multiple properties over multiple data streams, which introduces additional complexity and demands advanced optimization techniques. To address this challenge, we plan to incorporate well-known compiler optimization techniques such as common subexpression elimination and formula rewriting—into the construction process for multi-property monitors. Our compositional approach to monitor construction is particularly well-suited to supporting these optimizations in a modular and scalable manner. Ultimately, these enhanced monitoring capabilities, including multi-property handling and optimization, will be seamlessly integrated into our existing cloud-native infrastructure. This will make our approach more adaptable to real-world deployment scenarios.

REFERENCES

- [ACM97] Eugene Asarin, Paul Caspi, and Oded Maler. A Kleene theorem for timed automata. In *Proceedings of the Symposium on Logic in Computer Science (LICS)*, pages 160–171, 1997.
- [AH92] Rajeev Alur and Thomas A Henzinger. Logics and models of real time: A survey. In *Proceedings of the REX Workshop on Real-Time: Theory in Practice*, pages 74–106, 1992.
- [AH94] Rajeev Alur and Thomas A Henzinger. A really temporal logic. *Journal of the ACM (JACM)*, 41(1):181–203, 1994.

- [BBKT05] Saddek Bensalem, Marius Bozga, Moez Krichen, and Stavros Tripakis. Testing conformance of real-time applications by automatic generation of observers. *Electronic Notes in Theoretical Computer Science*, 113:23–43, 2005.
- [BDD⁺18] Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Ničković, and Sriram Sankaranarayanan. Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. *Lectures on Runtime Verification: Introductory and Advanced Topics*, pages 135–175, 2018.
- [BDH⁺22] David Basin, Thibault Dardinier, Nico Hauser, Lukas Heimes, Jonathan Julián Huerta y Munive, Nicolas Kaltensch, Srđan Krstić, Emanuele Marsicano, Martin Raszyk, Joshua Schneider, et al. Verimon: A formally verified monitoring tool. In *International Colloquium on Theoretical Aspects of Computing*, pages 1–6. Springer, 2022.
- [BG92] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992.
- [BHKZ11] David Basin, Matúš Harvan, Felix Klaedtke, and Eugen Zălinescu. Monpoly: Monitoring usage-control policies. In *International Conference on Runtime Verification (RV)*, pages 360–364. Springer, 2011.
- [BKMZ15] David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. Monitoring metric first-order temporal properties. *Journal of the ACM*, 62(2):15, 2015.
- [BKT17] David Basin, Srdjan Krstić, and Dmitriy Traytel. Aerial: Almost event-rate independent algorithms for monitoring metric regular properties. In *RV-CuBES*, pages 29–36, 2017.
- [BKZ17] David Basin, Felix Klaedtke, and Eugen Zălinescu. The MonPoly monitoring tool. In *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*, volume 3, pages 19–28, 2017.
- [BKZ18] David Basin, Felix Klaedtke, and Eugen Zălinescu. Algorithms for monitoring real-time properties. *Acta Informatica*, 55(4):309–338, 2018.
- [BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):1–64, 2011.
- [BN13] Kevin Baldor and Jianwei Niu. Monitoring dense-time, continuous-semantics, metric temporal logic. In *Runtime Verification: Third International Conference, RV 2012, Istanbul, Turkey, September 25–28, 2012, Revised Selected Papers 3*, pages 245–259. Springer, 2013.
- [CHL⁺18] Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. Tessla: temporal stream-based specification language. In *Formal Methods: Foundations and Applications: 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26–30, 2018, Proceedings 21*, pages 144–162. Springer, 2018.
- [CHR91] Zhou Chaochen, Charles Anthony Richard Hoare, and Anders P Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.
- [CM20] Agnishom Chattopadhyay and Konstantinos Mamouras. A verified online monitor for metric temporal logic with quantitative semantics. In *Runtime Verification: 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6–9, 2020, Proceedings 20*, pages 383–403. Springer, 2020.
- [CPS09a] Christian Colombo, Gordon J Pace, and Gerardo Schneider. Larva—safer monitoring of real-time java programs (tool paper). In *2009 seventh ieee international conference on software engineering and formal methods*, pages 33–37. IEEE, 2009.
- [CPS09b] Christian Colombo, Gordon J Pace, and Gerardo Schneider. Safe runtime verification of real-time properties. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 103–117. Springer, 2009.
- [DHF14] Adel Dokhanchi, Bardh Hoxha, and Georgios Fainekos. On-line monitoring for temporal logic robustness. In *International Conference on Runtime Verification*, pages 231–246. Springer, 2014.
- [Dru06] Doron Drusinsky. On-line monitoring of metric temporal logic with time-series constraints using alternating finite automata. *Journal of Universal Computer Science*, 12(5):482–498, 2006.
- [dSS⁺05] Ben d’Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B Sipma, Sandeep Mehrotra, and Zohar Manna. Lola: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME’05)*, pages 166–174. IEEE, 2005.

[EFH⁺03] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In *Proceedings of the Conference on Computer Aided Verification (CAV)*, pages 27–39, 2003.

[FFS⁺19] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah. Streamlab: stream-based monitoring of cyber-physical systems. In *International Conference on Computer Aided Verification*, pages 421–431. Springer, 2019.

[FK09] Bernd Finkbeiner and Lars Kuhtz. Monitor circuits for ltl with bounded and unbounded future. In *International Workshop on Runtime Verification*, pages 60–75. Springer, 2009.

[FMNP19] Thomas Ferrere, Oded Maler, Dejan Nićković, and Amir Pnueli. From real-time logic to timed automata. *Journal of the ACM (JACM)*, 66(3):1–31, 2019.

[Gab89] Dov Gabbay. The declarative past and imperative future: Executable temporal logic for interactive systems. In *Temporal Logic in Specification: Altrincham, UK, April 8–10, 1987 Proceedings*, pages 409–448. Springer, 1989.

[GS18] Felipe Gorostiaga and César Sánchez. Striver: Stream runtime verification for real-time event-streams. In *International Conference on Runtime Verification*, pages 282–298. Springer, 2018.

[HCRP91] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[HJL03] John Håkansson, Bengt Jonsson, and Ola Lundqvist. Generating online test oracles from temporal logic specifications. *International Journal on Software Tools for Technology Transfer*, 4:456–471, 2003.

[HKO⁺23] Klaus Havelund, Panagiotis Katsaros, Moran Omer, Doron Peled, and Anastasios Temperekidis. Tp-dejavu: Combining operational and declarative runtime verification. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 249–263. Springer, 2023.

[HOW14] Hsi-Ming Ho, Joël Ouaknine, and James Worrell. Online monitoring of metric temporal logic. In *International Conference on Runtime Verification*, pages 178–192. Springer, 2014.

[HPU17] Klaus Havelund, Doron Peled, and Dogan Ulus. First-order temporal logic monitoring with BDDs. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2017.

[HPU18] Klaus Havelund, Doron Peled, and Dogan Ulus. Dejavu: A monitoring tool for first-order temporal logic. In *Proceedings of the Workshop on Monitoring and Testing of Cyber-Physical Systems (MT-CPS)*, pages 12–13, 2018.

[HR04] Klaus Havelund and Grigore Roşu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer*, 6(2):158–173, 2004.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[LHR⁺23] Leonardo Lima, Andrei Herasimau, Martin Raszyk, Dmitriy Traytel, and Simon Yuan. Explainable online monitoring of metric temporal logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 473–491. Springer, 2023.

[MCW23] Konstantinos Mamouras, Agnishom Chattopadhyay, and Zhifu Wang. A compositional framework for algebraic quantitative online monitoring over continuous-time signals. *International Journal on Software Tools for Technology Transfer*, pages 1–17, 2023.

[MN13] Oded Maler and Dejan Nićković. Monitoring properties of analog and mixed-signal circuits. *International Journal on Software Tools for Technology Transfer*, 15(3):247–268, 2013.

[MNP05] Oded Maler, Dejan Nićković, and Amir Pnueli. Real time temporal logic: Past, present, future. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 2–16. Springer, 2005.

[MNP06] Oded Maler, Dejan Nićković, and Amir Pnueli. From MITL to timed automata. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 274–289. Springer, 2006.

[Mos84] Ben Moszkowski. Executing temporal logic programs. In *Proceedings of the Conference on Concurrency Theory (CONCUR)*, pages 111–130, 1984.

[MS03] Nicolas Markey and Philippe Schnoebelen. Model checking a path. In *Proceedings of the Conference on Concurrency Theory (CONCUR)*, pages 251–265, 2003.

- [MW20] Konstantinos Mamouras and Zhifu Wang. Online signal monitoring with bounded lag. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3868–3880, 2020.
- [NLM⁺18] Dejan Ničković, Olivier Lebeltel, Oded Maler, Thomas Ferrère, and Dogan Ulus. AMT2.0: Qualitative and quantitative trace analysis with extended signal temporal logic. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2018.
- [PGMN10] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A hard real-time runtime monitor. In *International Conference on Runtime Verification*, pages 345–359. Springer, 2010.
- [Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, 1977.
- [Ran58] George N Raney. Sequential functions. *Journal of the ACM*, 5(2):177–180, 1958.
- [Rey16] Mark Reynolds. Metric temporal logic revisited. *Acta Informatica*, 53(3):301–324, 2016.
- [RFB14] Thomas Reinbacher, Matthias Függer, and Jörg Brauer. Runtime verification of embedded real-time systems. *Formal methods in system design*, 44(3):203–239, 2014.
- [SBKT19] Joshua Schneider, David Basin, Srđan Krstić, and Dmitriy Traytel. A formally verified monitor for metric first-order temporal logic. In *Runtime Verification: 19th International Conference, RV 2019, Porto, Portugal, October 8–11, 2019, Proceedings 19*, pages 310–328. Springer, 2019.
- [SSA⁺19] César Sánchez, Gerardo Schneider, Wolfgang Ahrendt, Ezio Bartocci, Domenico Bianculli, Christian Colombo, Yliès Falcone, Adrian Francalanza, Srđan Krstić, Joao M Lourenço, et al. A survey of challenges for runtime verification from advanced application domains (beyond software). *Formal Methods in System Design*, 54:279–335, 2019.
- [TR05] Prasanna Thati and Grigore Roşu. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science*, 113:145–162, 2005.
- [UFAM14] Dogan Ulus, Thomas Ferrère, Eugene Asarin, and Oded Maler. Timed pattern matching. In *Proceedings of the Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, pages 222–236, 2014.
- [UFAM16] Dogan Ulus, Thomas Ferrère, Eugene Asarin, and Oded Maler. Online timed pattern matching using derivatives. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 736–751, 2016.
- [Ulu19] Dogan Ulus. Timescales: A benchmark generator for MTL monitoring tools. In *Proceedings of the Conference on Runtime Verification (RV)*, pages 402–412. Springer, 2019.
- [UM18] Dogan Ulus and Oded Maler. Specifying timed patterns using temporal logic. In *Proceedings of the Conference on Hybrid Systems: Computation and Control (HSCC)*, 2018.
- [vB91] Johan van Benthem. *The Logic of Time. A Model-Theoretic Investigation into the Varieties of Temporal Ontology and Temporal Discourse*. Springer, 1991.