Online Monitoring of Metric Temporal Logic using Sequential Networks

Dogan Ulus
Boston University
Boston, MA, USA
doganulus@gmail.com

ABSTRACT

Metric Temporal Logic (MTL) is a popular formalism to specify patterns with timing constraints over the behavior of cyber-physical systems. In this paper, we propose sequential networks for online monitoring applications and construct network-based monitors from the past fragment of MTL over discrete and dense time behaviors. This class of monitors is more compositional, extensible, and easily implementable than other monitors based on rewriting and automata. We first explain our sequential network construction over discrete time behaviors and then extend it towards dense time by adopting a point-free approach. Our formulation for dense time behaviors and MTL radically differs from the traditional pointy definitions and in return we avoid some longstanding complications. We argue that the point-free approach is more natural and practical therefore should be preferred for dense time applications. Finally, we present our implementation together with some experimental results that illustrate the performance of our monitors compared to similar existing tools.

CCS CONCEPTS

• **Theory of computation → Timed and hybrid models**; **Modal and temporal logics**;

ACM Reference Format:
Dogan Ulus. 2019. Online Monitoring of Metric Temporal Logic using Sequential Networks. In *Proceedings of 22st International Conference on Hybrid Systems: Computation and Control (part of CPS Week) (HSCC '19).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/xxxxxxx.xxxxxxx

1 INTRODUCTION

Monitoring temporal behaviors of systems during their actual execution has important application areas ranging from runtime verification and anomaly detection to supervisory control. It is desirable to construct such online monitors automatically from high-level specifications of temporal patterns. Originally proposed for formal verification, the formalism of Linear Temporal Logic (LTL) [19] and its timed extensions such as Metric Temporal Logic (MTL) [14] are very popular to specify various temporal patterns to be monitored. In this paper, we propose online monitors based on sequential networks and study their construction from the past fragment of MTL over discrete and dense time behaviors.

A sequential network is an abstract machine defined by a finite state space, a set of update rules for each state variable, and an output function. It operates by reading a sequence (temporal behavior) left-to right (past-to-future), updates its state variables according to some certain equations, and yields an output value at each step. Hence, it is easy to view sequential networks as an analogue of dynamical systems in theoretical computer science. Sequential networks are functionally equivalent to finite automata but they differ in the structure. Such seemingly subtle difference brings significant advantages for sequential networks in terms of compositionality, extensibility, and implementability. Therefore, sequential networks constitute a very solid foundation to meet various types of monitoring needs in practice.

We construct sequential networks directly from the past fragment of MTL specifications. The restriction to the past temporal operators is twofold: (1) Future-oriented (acausal) monitoring is inherently more expensive than past-oriented (causal) monitoring. The worst-case exponential cost of bookkeeping all possibilities in the future cannot be avoided unless the output at t is delayed by some amount 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 take a timely decision as in the supervisory control systems. Moreover, a monitor that frequently returns *unknown* or *pending* as answers is not helpful in such applications. Therefore, we consider future temporal operators as a costly feature without much practical benefit in online monitoring applications.

The essence of our sequential network construction from temporal logic formulas is to associate each subformula with a state variable to store relevant information and then manipulate them using a suitable algebra. Here we will start from the online monitor construction explained in [12] where authors essentially construct Boolean sequential networks from past LTL formulas. For monitoring temporal patterns with timing constraints, the Boolean algebra alone would no longer suffice in sequential networks unlike the case of untimed patterns. Therefore, sequential networks to monitor MTL formulas over discrete and dense time behaviors needs to store numerical timing information. Our technique in this paper differs from similar works [5] as we do not explicitly store sets of past time points/periods from where timing constraints will be measured. Instead we store and manipulate sets of future time

This paper is dedicated to the memory of Dr. Oded Maler.

HSCC '19, April 16–18, 2019, Montreal, QC, Canada

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.1145/xxxxxxx.xxxxxxx

points/periods affected by temporal operators based on an observation [15] for offline MTL monitoring. Furthermore we emphasize sequential networks as the model of computation for monitoring tasks. This approach provides further connections with the theory of automata and some related theoretical results. Effectiveness of these techniques then would come no surprise for anyone despite their simplicity.

The other contribution of this paper is in introducing an alternative (point-free) dense time formulation for MTL based on time periods rather than time points. Our motivation is that a number of (ancient and modern) complications arises when we extend the discrete time setting towards the dense time if dense timeline has been considered to be an infinite collection (continuum) of time points. It is known that these mathematical complications simply disappear once the traditional pointy definition of timeline has been abandoned. Recently a number of works has proposed a point-free approach to monitor timed regular expressions and interval logic [21, 22], which are based on time periods by definition. Following these works, the question whether an inherently pointy temporal logic such as MTL can be defined and monitored effectively without points becomes interesting. In this paper, we give a positive answer for this question and provide necessary foundations to have a natural and practical dense time monitoring of MTL without longstanding complications of the continuum and ever-increasing number of patches to fix them.

The structure of the paper is as follows. We first give the necessary background and definitions in Section 2. Sequential networks constructions from Past LTL and MTL specifications over discrete time behaviors is explained with examples in Section 3. This section also serves as a preparation for dense time constructions given in Section 4. We then formalize our dense time setting at the end of Section 4. Our implementation[1] and evaluation results are presented in Section 5 before the conclusions.

2 DEFINITIONS

2.1 Temporal Behaviors and Logics

Let $P = \{p_1, \ldots, p_m\}$ be a finite set of propositions that correspond to some qualitative states and activities of some real-time systems and the environment. We consider such propositions to be observed incrementally for an indefinite amount of time. Then a (Boolean) temporal behavior refers to the history of such observations over either discrete or dense timeline. More precisely, we first define an alphabet $\Sigma = \mathbb{B}^m$ of observations expressed as Boolean vectors of dimension m where $\mathbb{B} = \{\mathtt{F}, \mathtt{T}\}$. Then discrete time behaviors are given as functions from integers to Boolean vectors. Similarly dense time behaviors are traditionally defined to be functions from real or rational numbers to Boolean vectors.

Past Linear-time Temporal Logic. Past Linear-time Temporal Logic (past LTL) extends the propositional logic with the *previously* ($\bullet$) and *since* ($\mathcal{S}$) operators. Given a finite set P of atomic propositions, the formulas of past LTL are built using the following grammar:

$$\varphi \;=\; \top \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bullet\varphi \mid \varphi_1 \,\mathcal{S}\, \varphi_2$$

[1] Available for the repeatability evaluation

The satisfaction relation $(w,t) \vdash \varphi$ of past LTL indicates that the discrete-time behavior w satisfies φ at the time point t.

$$\begin{array}{lll}
(w,t) \vdash p & \leftrightarrow & w_p(t) = \top \\
(w,t) \vdash \neg\varphi & \leftrightarrow & (w,t) \nvdash \varphi \\
(w,t) \vdash \varphi_1 \vee \varphi_2 & \leftrightarrow & (w,t) \vdash \varphi_1 \text{ or } (w,t) \vdash \varphi_2 \\
(w,t) \vdash \bullet\varphi & \leftrightarrow & (w,t-1) \vdash \varphi \\
(w,t) \vdash \varphi_1 \,\mathcal{S}\, \varphi_2 & \leftrightarrow & \exists t' \in (0,t].\ (w,t') \vdash \varphi_2 \text{ and} \\
 & & \quad \forall t'' \in (t',t].\ (w,t'') \vdash \varphi_1
\end{array}$$

Other popular temporal operators *past eventually* ($\blacklozenge$) and *past always* ($\blacksquare$) can be obtained from the since operator using equivalences $\blacklozenge\varphi \equiv \top\mathcal{S}\varphi$ and $\blacksquare\varphi \equiv \neg\blacklozenge\neg\varphi$.

Past Metric Temporal Logic. Past Metric Temporal Logic (past MTL) extends propositional logic with a time-bounded (timed) variant of the *since* modality, denoted by $\mathcal{S}_I$, such that an interval I of duration values additionally restricts the range of quantification over time points. Given a finite set P of atomic propositions, the formulas of past MTL, when interpreted over discrete-time behaviors, are defined by the following grammar:

$$\varphi \;=\; \top \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \,\mathcal{S}_I\, \varphi_2$$

where $p \in P$. Time-bounded *past eventually* ($\blacklozenge_I$) and *past always* ($\blacksquare_I$) are derived as usual. For any temporal operator, we omit the time bound I if $I = [0, \infty)$ and call such a operator untimed. The traditional satisfaction relation $(w,t) \vdash \varphi$ indicates that the sequence w satisfies the formula φ at time point $t > 0$ as follows.

$$\begin{array}{lll}
(w,t) \vdash p & \leftrightarrow & w_p(t) = \top \\
(w,t) \vdash \neg\varphi & \leftrightarrow & (w,t) \nvdash \varphi \\
(w,t) \vdash \varphi_1 \vee \varphi_2 & \leftrightarrow & (w,t) \vdash \varphi_1 \text{ or } (w,t) \vdash \varphi_2 \\
(w,t) \vdash \varphi_1 \,\mathcal{S}_I\, \varphi_2 & \leftrightarrow & \exists t' \in (0,t).\ (w,t') \vdash \varphi_2 \text{ and} \\
 & & \quad \forall t'' \in (t',t).\ (w,t'') \vdash \varphi_1 \text{ and} \\
 & & \quad\quad t - t' \in I
\end{array}$$

2.2 Sequential Networks

A sequential network is an abstract machine that computes an output sequence $Y_1 Y_2 \ldots Y_k \ldots$ from an input sequence $X_1 X_2 \ldots X_k \ldots$. At any time step k, a sequential network is in a state, denoted by V_k, given by the values of its state variables. The network operates by updating its state V_k and yielding the current output Y_k with respect to the previous state V_{k-1} and the current input X_k. We then completely characterize a sequential network by the following elements:

- The initial valuation vector V_0 of m state variables
- Update equations $V_k(i) = F_i(V_{k-1}, X_k)$ for $i = 1, \ldots, m$
- The output function $Y(V_{k-1}, X_k)$ of the network

Figure 1 illustrates the model of computation employed in sequential networks. Easily seen that we only need to maintain a fixed set of state variables during a run of the network and the number of updates linearly depends on the number of state variables and the length of the input. We can usually afford update functions that are expensive in terms of complexity as long as their complexities do not depend on the length of the input (or depends on logarithmically at most). Actual data types of inputs, outputs, and state variables of the network vary across applications. The

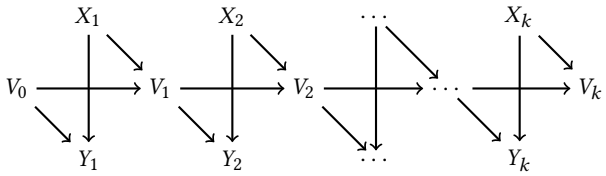

Figure 1: Computation in sequential networks.

B	$V_k(\neg r)$	$=$	$\neg X_k(r)$
B	$V_k(p \vee q)$	$=$	$X_k(p) \vee X_k(q)$
B	$V_k(\varphi)$	$=$	$V_k(\neg r) \ \vee \ \Big(V_k(p \vee q) \wedge V_{k-1}(\varphi)\Big)$
B	Y_k	$=$	$V_k(\varphi)$

Table 1: Boolean sequential network constructed from the formula $\varphi = (p \vee q) \ \mathcal{S} \ \neg r$.

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 such as flip-flops. This class of networks precisely recognize regular languages and thus functionally equivalent to finite automata [8]. However, there are a few key differences between automata and networks regarding the direction of updates explained as follows. While the transition function of an automaton computes the next reachable states from a state (one-to-many), an update rule of a sequential network computes whether a state is reachable from previous states (many-to-one). We refer to the study [7] for some theoretical results regarding this reverse relation between automata and Boolean sequential networks.

Finally, update equations and the output function in sequential networks are given as functions of previous valuations V_{k-1} of state variables in the strict sense. However, in this paper, we allow so-called transient state variables to ease the forthcoming constructions. These transient state variables store intermediate results of update computations and depend on current valuations V_k. If desired, such states can be eliminated by embedding their update equations into the rest as a small optimization attempt.

3 DISCRETE TIME

In this section, we explain sequential network constructions over the discrete time behaviors from past LTL and MTL specifications. In discrete setting, a fixed set of propositions is observed at each discrete time point and we feed the network with the corresponding Boolean vector. For the case of MTL, we additionally assume a counter that keeps track of the integer time (sequence index) and use integers to specify timing constraints accordingly.

3.1 Sequential Networks from Past LTL

Our monitor construction from past LTL is very similar to the one presented in [12] introduced under the broad name of dynamic programming. Here we show that this algorithm essentially constructs Boolean sequential networks from past LTL specifications. Although it is mostly a change in terminology, we emphasize some elements that are trivial for past LTL monitors (such as output functions) as a preparation for timed generalizations in the following.

Monitor construction from a past LTL formula starts by associating each subformula with a Boolean state variable initially set to false. Update equations for the Boolean state variables are defined according to their operators as given in the list below:

$$V_k(p) \quad = \quad X_k(p) = \begin{cases} 1 & \text{if } p \text{ holds at } k \\ 0 & \text{otherwise.} \end{cases}$$

$$\begin{aligned} V_k(\neg\varphi) &= \neg V_k(\varphi) \\ V_k(\varphi_1 \vee \varphi_2) &= V_k(\varphi_1) \vee V_k(\varphi_2) \\ V_k(\bullet\, \varphi) &= V_{k-1}(\varphi) \\ V_k(\varphi_1 \mathcal{S} \varphi_2) &= V_k(\varphi_2) \ \vee \ \big(V_k(\varphi_1) \wedge \ V_{k-1}(\varphi_1 \mathcal{S} \varphi_2)\big) \end{aligned}$$

We then obtain update equations for other temporal operators using their definitions in terms of the since operator as follows.

$$\begin{aligned} V_k(\blacklozenge\, \varphi) &= V_k(\varphi) \vee V_{k-1}(\blacklozenge\, \varphi) \\ V_k(\blacksquare\, \varphi) &= V_k(\varphi) \wedge V_{k-1}(\blacksquare\, \varphi) \end{aligned}$$

The construction is completed by declaring the value of the top variable (the original formula) as the output of the overall network.

We now illustrate the sequential network construction from past LTL with an example. Consider a past LTL formula $\varphi = (p \vee q) \ \mathcal{S} \ \neg r$, which contains three propositions and three subformulas. We then construct a sequential network from φ, which has three inputs and three state variables (all initialized to false) with update equations and output function listed in Table 1. Letters in the leftmost column of the table denotes datatype of variables where B stands for Boolean, which is the case for untimed monitors over discrete time behaviors.

Note that compositionality is an important feature of network based monitors. Easily seen, we can construct the same sequential network from previously constructed monitors of $(p \vee q)$ and $\neg r$. Given two monitors for past LTL formulas $\varphi_1 = (p \vee q)$ and $\varphi_2 = \neg r$, we can obtain a new monitor for $\varphi_1 \ \mathcal{S} \ \varphi_2$ 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 $\varphi_1 \ \mathcal{S} \ \varphi_2$ at the composition. Notice that output equations defined for past LTL are trivial, which simply yield the value of the corresponding state variable. Therefore a discussion of output equations can be ignored for past LTL constructions. However, this is not the case for constructions from MTL as we see in the following.

3.2 Sequential Networks from Past MTL

For monitoring timed specifications such as past MTL, we quantitatively reason about time distances (durations) between events and states. For example, consider the evaluation of a MTL formula

$\varphi = \varphi_1 \mathcal{S}_{[a,b]} \varphi_2$ at the current time point k. According to the semantics, we need to check distances between k and time points where φ_2 hold in the past whether they satisfy the specified timing constraint $[a, b]$. In other words, actual time points where φ_2 hold in the past are important for the MTL evaluation and we need to store relevant time points (thus a set of integers) during monitoring. Compare this with the untimed case where the existence of such a time point needs to be stored (thus a Boolean value). Therefore, in the following, we employ state variables that store sets of integers, called timed state variables, to handle quantitative timing constraints. Unless explicitly stated, we consider all timed state variables are initialized to the empty set $\emptyset$. Then it is sufficient to define update equations and the output function for each timed temporal operator to manipulate such sets according to the semantics. Note that Boolean and untimed temporal operators in MTL formulas can be handled using Boolean state variables as if they are LTL formulas.

Timed Past Eventuality Operator. According to (discrete time) MTL semantics, the formula $\varphi = \blacklozenge_{[a,b]} \psi$ holds at a time point k if the subformula ψ holds for some time points in $[k-b, k-a]$. Equivalently, whenever the formula ψ holds at a time point k, the formula φ will be true for all future time points in $[k+a, k+b]$. Using this simple observation, called forward-shifting in [15], we first define a state variable $V_k(\psi) \subseteq [k, \infty)$ for the formula $\varphi = \blacklozenge_{[a,b]} \psi$ to store a set of integers. Intuitively, the set $V_k(\blacklozenge_{[a,b]})$ corresponds to future time points at which the formula $\blacklozenge_{[a,b]}$ will be held. Therefore, we first define the update equation for $V_k(\varphi)$ as follows.

$$V_k(\blacklozenge_{[a,b]}\psi) = \begin{cases} V_{k-1}(\blacklozenge_{[a,b]}\psi) \cup [k+a, k+b] & \text{if } Y_k(\psi) \\ V_{k-1}(\blacklozenge_{[a,b]}\psi) & \text{otherwise.} \end{cases}$$

where the function $Y_k(\psi)$ denotes the output equation of the network constructed for ψ. Note that ensuring the condition $V_k \subseteq [k, \infty)$ is important to bound the size of the set $V_k(\varphi)$. Hence, we also apply the intersection $V_k(\varphi) \cap [k, \infty)$ at each update. For brevity, we omit this intersection operation in all the following update equations.

It is easy to see that the current time point k must be in $V_k(\blacklozenge_{[a,b]}\psi)$ by definition if the formula $\blacklozenge_{[a,b]}\psi$ holds at k. Therefore, we define the output function $Y_k(\blacklozenge_{[a,b]}\psi)$ to be a membership test as follows.

$$Y_k(\blacklozenge_{[a,b]}\psi) = k \in V_k(\blacklozenge_{[a,b]}\psi)$$

We now illustrate the sequential network construction from a past MTL formula $\varphi = \blacklozenge_{[1,2]}\blacklozenge_{[1,2]}(p \vee q)$, which possesses (non-propositional) subformulas (1) : $p \vee q$, (2) : $\blacklozenge_{[1,2]}(p \vee q)$, and (3) : $\blacklozenge_{[1,2]}\blacklozenge_{[1,2]}(p \vee q)$. Accordingly we have three state variables in the network, one Boolean and two timed, as shown in Table 2 with update equations and the output function. Notice that the output function $k \in V_k(2)$ of $\blacklozenge_{[1,2]}(p \vee q)$ is embedded into the update equation of $V_k(3)$ at the composition.

In Table 3, we depict a discrete time behavior from the index $k = 0$ to $k = 6$ over the propositions p and q by listing on separate lines. The next three lines below denotes the valuation of state variables and the final line denotes the output of the sequential network. We note that the formula $\blacklozenge_{[1,2]}\blacklozenge_{[1,2]}(p \vee q)$ is equivalent to a simpler formula of $\blacklozenge_{[2,4]}(p \vee q)$. Such equivalences and rewrites can be used to optimize the sequential network construction but it is important for us to have a robust construction not depending on these optimizations.

B	$V_k(1)$	=	$X_k(p) \vee X_k(q)$	
T	$V_k(2)$	=	$\begin{cases} V_{k-1}(2) \cup [k+1, k+2] \\ V_{k-1}(2) \end{cases}$	if $V_k(1)$ otherwise.
T	$V_k(3)$	=	$\begin{cases} V_{k-1}(3) \cup [k+1, k+2] \\ V_{k-1}(3) \end{cases}$	if $k \in V_k(2)$ otherwise.
B	$Y_k(\varphi)$	=	$k \in V_k(3)$	

Table 2: Sequential network constructed from the formula $\varphi = \blacklozenge_{[1,2]}\blacklozenge_{[1,2]}(p \vee q)$.

k :	0	1	2	3	4	5	6
p :		T	F	F	F	F	F
q :		F	F	F	F	T	F
$V_\varphi(1)$:	F	T	F	F	F	T	F
$V_\varphi(2)$:	$\emptyset$	$[2,3]$	$[2,3]$	$\{3\}$	$\emptyset$	$[6,7]$	$[6,7]$
$V_\varphi(3)$:	$\emptyset$	$\emptyset$	$[3,4]$	$[3,5]$	$[4,5]$	$\{5\}$	$[7,8]$
Y_φ :		F	F	T	T	T	F

Table 3: A run of the network in Table 2.

Finally a key point of our construction is in representing sets of integers by symbolic means, that is to say, as a union of intervals. By this way, large numbers (and infinity) that may appear in timing constraints can be handled efficiently. Set operations between an interval and union of intervals (as we need in our update equations) and membership tests can be performed in (amortized) logarithmic time. The worst case space requirement can be observed for the formula $\blacklozenge_{[b,b]}\psi$ where b is a finite number and ψ holds at every other time point. In this case, we need to store $b/2$ singleton intervals (time points) in $V_k(\blacklozenge_{[b,b]}\psi)$.

Timed Past Always Operator. The formula $\varphi = \blacksquare_{[a,b]} \psi$ holds at a time point t if the subformula ψ holds for all time points in $[t-b, t-a]$. We can obtain the update equation

$$V_k(\varphi) = \begin{cases} V_{k-1}(\varphi) \cup [k+a, k+b] & \text{if not } Y_k(\psi) \\ V_{k-1}(\varphi) & \text{otherwise.} \end{cases}$$

and the output equation

$$Y_k(\varphi) = k \notin V_k(\varphi)$$

from the case of timed past eventuality operator as usual. We illustrate an example run of the sequential network constructed from the formula $\varphi = \blacksquare_{[1,2]}\psi$ in Table 4.

Notice that the network yields true for time point 1 as there is no time points before time point 1; therefore, it holds vacuously. This is the standard behavior equivalent to the assumption that ψ holds for all time point before time point 1. Alternatively, we can invoke strong semantics of the always operator and assume the formula ψ does not hold before time point 1. Under strong semantics, we

$k:$	0	1	2	3	4	5	6
$Y_\psi:$		F	F	T	T	T	F
$V(\varphi):$	$\emptyset$	$[2,3]$	$[2,4]$	$[3,4]$	$\{4\}$	$\emptyset$	$[7,8]$
$Y_\varphi:$		T	F	F	F	T	T

Table 4: A run of the network for $\varphi = \blacksquare_{[1,2]}\psi$.

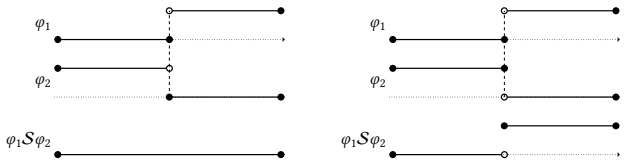

Figure 2: The outcome of monitoring $\varphi_1 S \varphi_2$ changes due to the formula φ_2 holds zero duration more on the right.

simply set the initial valuation $V_0 = [0, b]$ where b is the upper time bound b for the operator.

Timed Since Operator. The formula $\psi_1 \mathcal{S}_{[a,b]} \psi_2$ holds at a time point t if ψ_2 held at time point t' in the past between time points $t-b$ and $t-a$, and ψ_1 has held continuously from t' to t. Intuitively speaking, we can see these semantics as a conjunction of the case of timed past eventuality as we forward-shift time points at which φ_2 hold and the case where φ_1 continuously holds. It means that we can store time point V_k as if timed past eventuality while φ_1 is holding. On the other hand, time points in the set V_k becomes invalid whenever φ_1 ceases to hold therefore we need to clear V_k. Based on these observations, we define the update equation of the sequential network constructed for the formula $\varphi = \psi_1 \mathcal{S}_{[a,b]} \psi_2$ as follows.

$$V_k(\varphi) = \begin{cases} V_{k-1}(\varphi) \cup [k+a, k+b] & \text{if } Y_k(\psi_1) \wedge Y_k(\psi_2) \\ [k+a, k+b] & \text{if } \neg Y_k(\psi_1) \wedge Y_k(\psi_2) \\ V_{k-1}(\varphi) & \text{if } Y_k(\psi_1) \wedge \neg Y_k(\psi_2) \\ \emptyset & \text{otherwise.} \end{cases}$$

The output equation is similarly defined.

$$Y_k(\varphi) = k \in V_k(\varphi)$$

The Table 5 illustrates a run of the sequential network constructed from the formula $\varphi = p \; \mathcal{S}_{[2,3]} \; q$.

$k:$	0	1	2	3	4	5	6
$p:$		F	F	T	T	T	T
$q:$		F	T	F	F	T	F
$V(\varphi):$	$\emptyset$	$\emptyset$	$[4,5]$	$[4,5]$	$[4,5]$	$\{5\} \cup [7,8]$	$[7,8]$
$Y(\varphi):$		F	F	F	T	T	F

Table 5: A run of the sequential network constructed for the formula $p \; \mathcal{S}_{[2,3]} \; q$.

4 DENSE TIME WITHOUT POINTS

In this section, we explain sequential network constructions from past MTL specifications over dense time behaviors. Under the hood, we propose an alternative (point-free) dense-time formulation for MTL based on time periods to replace the traditional formulation based on time points. Our goal has been to avoid longstanding complications of the traditional pointy model of time and consequently obtain a more natural and practical MTL monitoring solution over dense time behaviors without sacrificing formalness.

Traditionally, dense timeline has been viewed as an infinite collection (continuum) of time points having zero duration. This view is so rooted that the resulting complications as well as patches to fix them are now thought to be the fate of dense timeline. Here we focus on two of them we have been experiencing in contemporary timed systems research. First, by the very definition of the continuum, it is *possible* to have a dense time behavior that switches infinite number times in a finite amount of time. Since this is very problematic in computer science regarding computability and complexity, virtually all practical works assume a (finite variability) condition that bounds the number of switchings or events in a finite amount of time. Here it is beneficial to question why we have started with a clearly non-physical assumption at the first place and then patched it with a physical restriction everywhere else. We think, if such a non-physical assumption is an idealization of physics (which is perfectly fine), then it simply should help solve our problems rather than create them. At this point, readers are encouraged to revise ancient complications [9] as well.

On the other hand, the second complication we want to show is much more annoying. Consider an (untimed) MTL formula $\varphi = \psi_1 \; \mathcal{S} \; \psi_2$ and its evaluation over pointy dense time behaviors of ψ_1 and ψ_2 depicted on the left and right of Figure 2. Although ψ_2 behaviors of both sides are the same everywhere except a single point, we obtain a different result when monitoring the formula $\psi_1 \; \mathcal{S} \; \psi_2$ according to the semantics of MTL. In other words, on the right of the figure, the formula ψ_2 held *precisely zero duration more* by including the time point at the boundary and consequently the outcome of monitoring has significantly altered. We argue that such an outcome totally defies the quantitative reasoning we try to have about time by distinguishing cases when there is no measurable difference even mathematically.

Note that it is also possible to patch this particular bug by restricting pointy behaviors to be left-continuous functions, that is to say, functions with no jumps when the limit point is approached from the left. Under the finite variability condition and left-continuity, the behavior can be then partitioned into a finite number of left-open and right-closed intervals. This solution would work in practice but see that this is yet another patch for the pointy model enforced externally. Patching after patching is never considered to be a good sign for such a fundamental theory after all. Besides notice that all these practical patches make the pointy model less pointy by introducing certain aspects of the point-free model such as finiteness and uniform parts of non-zero duration. Therefore we can say that the point-free model we are proposing here naturally emerges from the practice. Now let us move to our point-free proposal.

A point-free approach for time starts from an undivided timeline as a whole and assumes that it is the observer who divide it into parts during the observation. Since we cannot observe physical processes with an infinite precision and divide the dense timeline infinitely many times, we can only have a finite number of parts having non-zero durations (time periods). Such a view of time gives us a mathematical way to avoid Zeno's paradoxes [9] as well as our aforementioned complications. We, therefore, consider dense timeline to be a finite sequence of time periods rather than an infinite collection of time points in the following.

Point-free Behaviors. When we are describing temporal behaviors, it is natural to say, for example, that a proposition p is true for 4 seconds, then false for 2 seconds, then true for 2 seconds, and then false for 1 seconds. Equivalently, we can say the proposition is true from 0 to 4, false from 4 to 6, true from 6 to 8, and false from 8 to 9 assuming a beginning time zero. As time progress, we can observe further, divide the rest of the timeline, and say p is false from 9 to 11, and continue. These descriptions do not explicitly refer to values of individual time points but time periods. By point-free behaviors, we mean such finite sequences of time periods furnished with observation values.

Formally we define a point-free dense-time Boolean behavior w on a time period (t_0, t_n) to be a finite sequence such that

$$w = (t_0, t_1, b_1), (t_1, t_2, b_2), \ldots, (t_{n-1}, t_n, b_n)$$

where b_k is a Boolean value and $t_{k-1} < t_k$ for $k \in 1 \ldots n$. We then write the example proposition p as follows.

$$p = (0, 4, \text{T}), (4, 6, \text{F}), (6, 8, \text{T}), (8, 9, \text{F})$$

Notice that such representation is far from unique as we can divide any time period arbitrarily and obtain equivalent behaviors. For example, the behavior p' below is simply equivalent to the behavior p.

$$p' = (0, 2.5, \text{T}), (2.5, 4, \text{T}), (4, 6, \text{F}), (6, 8, \text{T}), (8, 9, \text{F})$$

More often we use the other direction and merge *stuttering* periods, that is successive time periods that have the same value, into one.

By the span of a behavior we mean the total time period it occupies on the timeline. For example, the behavior p has a span of $(0, 9)$. Once the span of a behavior is clear in the context, we drop periods of false from its representation and write periods of true in a set. For example, we represent the behavior p as a set $\{(0, 4), (6, 8)\}$ with a span of $(0, 9)$.

One practical benefit of the point-free approach is that we now have a single type of time periods in sequences. Compare it with the fact that pointy behaviors (partitioned into a finite sequence of intervals under the finite variability condition) require four types of intervals (open, closed, and half-opens) in sequences, which means more resources and effort for implementation and testing.

Boolean Operations. We first explain Boolean operations over point-free behaviors. For example, consider two point-free behaviors of propositions p_1 and p_2 spanning over a time period $(0, 20)$.

$$\begin{aligned} p &= \{(2, 4), (7, 10), (11, 17)\} \\ q &= \{(3, 8), (14, 15)\} \end{aligned}$$

Informally speaking, a Boolean operation over point-free behaviors amounts to synchronizing two behaviors (by dividing them accordingly), apply the usual Boolean operation over values, and merge stuttering periods. The following illustrates the results of some Boolean operations over behaviors p and q.

$$\begin{aligned} \sim p &= \{(0, 2), (4, 7), (10, 11), (17, 20)\} \\ p \sqcap q &= \{(3, 4), (7, 8), (14, 15)\} \\ p \sqcup q &= \{(2, 10), (11, 17)\} \end{aligned}$$

where operators $\sim$, $\sqcap$, and $\sqcup$ denote negation, conjunction, and disjunction over point-free behaviors, respectively. Notice that we compute the negation of a behavior with respect to its span unless specified otherwise. These operations are very intuitive and we later formalize them in the corresponding section.

For online monitoring, we assume we receive dense time behaviors one chunk at a step, which is called the current chunk or kth chunk referring to the number of steps so far. It does not mean that values of every proposition (and formula) are necessarily constant in a chunk. The reason for this is that we would like to stay faithful to the segmentation of the timeline given by the input behavior for the output behavior. Then, in general, constant chunks cannot be obtained in the presence of timed subformulas.

Next, similar to discrete time networks, we define state variables V_k and store point-free behaviors in state variables corresponding to subformulas of the original formula. For the case of Boolean operators, update equations are straightforward:

$$\begin{aligned} V_k(p) &= p_k \\ V_k(\neg\psi) &= \sim V_k(\psi) \\ V_k(\psi_1 \wedge \psi_2) &= V_k(\psi_1) \sqcap V_k(\psi_2) \\ V_k(\psi_1 \vee \psi_2) &= V_k(\psi_1) \sqcup V_k(\psi_2) \end{aligned}$$

where p_k is the kth chunk of the behavior p and all the valuations of state variables for Boolean operators span over the kth chunk. Output functions $Y_k(\varphi) = V_k(\varphi)$ are then trivially defined to be equal to corresponding state variable.

Timed Since Operation. In the case of the timed since, it is more convenient to compute the operation sequentially over time periods where both values of operands are constant. To this end, we locally introduce a secondary (and local) index l that indicates the position of a time period in a chunk, in which the values of operands are necessarily constant. Suppose we are processing the kth chunk that consists of n constant-valued time periods for the operands $Y_k(\varphi_1)$ and $Y_k(\varphi_2)$ of the since operation for the formula $\varphi_1 \mathcal{S}_{(a,b)} \varphi_2$. Note that the number n is specific to this chunk and this formula and we do not use it to divide the timeline for other operations in the network. We denote by $V_{k,l}$ the valuation of l-th constant time period for the both operand in the kth chunk and similarly by $Y_{k,l}$ the output of thereof for $l = 1 \ldots n$. Then we give update equations for the timed since operator $\psi = \varphi_1 \mathcal{S}_{(a,b)} \varphi_2$ as follows.

$$V_{k,l}(\psi) = \begin{cases} V_{k,l-1}(\psi) \sqcup (t + a,\ t' + b) & \text{if } y_{k,l}(\varphi_1) \wedge y_{k,l}(\varphi_2) \\ (t' + a,\ t' + b) & \text{if } \neg y_{k,l}(\varphi_1) \wedge y_{k,l}(\varphi_2) \\ V_{k,l-1}(\psi) & \text{if } y_{k,l}(\varphi_1) \wedge \neg y_{k,l}(\varphi_2) \\ \emptyset & \text{otherwise.} \end{cases}$$

where $y_{k,l}(\varphi)$ is the Boolean value for the constant time period $Y_{k,l}(\varphi)$ and $V_{k,n} = V_{k+1,0}$. The output of each local step equals

$k:$	1	2	3	4
curr :	$(0,30)$	$(30,47)$	$(47,75)$	$(75,99)$
$\varphi_1:$	$\{(7,30)\}$	$\{(30,35),(39,47)\}$	$\{(47,49),(63,75)\}$	$\{(75,99)\}$
$\varphi_2:$	$\{(3,8)\}$	$\{(38,39)\}$	$\{(70,75)\}$	$\{(75,89)\}$
$\|\|:$	$\varphi_1\varphi_2$ $(0,3):$ F F $(3,7):$ F T $(7,8):$ T T $(7,30):$ T F	$\varphi_1\varphi_2$ $(30,35):$ T F $(35,38):$ F F $(38,39):$ F T $(39,47):$ T F	$\varphi_1\varphi_2$ $(47,49):$ T F $(49,63):$ F F $(63,70):$ T F $(70,75):$ T T	$\varphi_1\varphi_2$ $(75,89):$ T T $(89,99):$ T F
$V_\psi:$	$1:\ \emptyset$ $2:\{(25,31)\}$ $3:\{(25,32)\}$ $4:\{(25,32)\}$	$1:\{(30,32)\}$ $2:\ \emptyset$ $3:\{(57,63)\}$ $4:\{(57,63)\}$	$1:\ \emptyset$ $2:\ \emptyset$ $3:\ \emptyset$ $4:\{(88,99)\}$	$1:\{(88,113)\}$ $2:\{(89,113)\}$
$Y_\psi:$	$\{(25,30)\}$	$\{(30,32)\}$	$\emptyset$	$\{(88,99)\}$

Table 6: A run of the network constructed for the formula $\psi = \varphi_1\ \mathcal{S}_{(18,24)}\ \varphi_2$ over dense time behaviors of φ_1 and φ_2.

to the conjunction of the valuation and the current local step as follows.

$$Y_{k,l}(\varphi) = V_{k,l} \sqcap \mathrm{Span}(k,l)$$

We then collectively yield the outcome $Y_k(\varphi)$ of the whole chunk k as the disjunction of the outputs of local steps.

$$Y_k(\varphi) = Y_{k,1} \sqcup Y_{k,2} \sqcup \cdots \sqcup Y_{k,n}$$

Yielding a collective output as above eliminates the fragmentation created by the local synchronization and allows us to continue the computation toward upper nodes seamlessly.

In Table 6 we illustrate all these concepts and operations over dense time behaviors of subformulas φ_1 and φ_2. Input behaviors in the example are incrementally given in the form of 4 chunks and we locally synchronize them before the timed since operation. This process may lead to further divisions on the timeline in which both behaviors hold constant as shown in the row denoted by ||. For this example, chunks are divided into four except the latest one, which is divided into two. Then we apply our update rule in a sequential manner inside chunks and collectively yield the output of each chunk before moving to the next chunk.

Logical Characterization of Point-free Operations. The traditional semantics of MTL over dense time is pointy, that is to say, it assigns a truth value to each time point on the timeline for a given formula and behavior. Our goal here is to develop an alternative point-free semantics for MTL by formalizing Boolean and temporal operations over point-free behaviors intuitively explained in the previous sections. To this end, we redefine MTL as a temporal logic of time periods, which rather assigns a truth value to each *time period* in a two-dimensional temporal structure for given a formula and a point-free behavior.

We start from a whole timeline, typically $(0,\infty)$, and denote the set of all time periods over the timeline by Ω. For convenience, we assume the bounds of time periods and timing constraints are given as rational numbers in the following. The set of all periods on which a proposition p holds is called the valuation set $\mathcal{V}(p) \subseteq \Omega$ of the proposition p. Intuitively, if a proposition p holds over a time period, then p necessarily holds for every for all sub-periods thereof. This property is called homogeneity and captured by the formula

$$(t,t') \in \mathcal{V}(p) \longleftrightarrow \forall t < r < r' < t.\ (r,r') \in \mathcal{V}(p) \qquad \text{(HOM)}$$

For example, consider a point-free behavior $p = \{(2,4),(6,9)\}$, which means the proposition p holds from 2 to 4 and 6 to 9. Then this behavior formally represents a valuation set $\mathcal{V}(p) = \{(r,r') \mid 2 \le r < r' \le 4 \text{ or } 6 \le r < r' \le 9\}$ that contains every time period the proposition p holds. In the following, we use point-free behaviors and homogeneous valuation sets interchangeably.

Similar to other temporal formalisms, we formalize MTL inside a (two-dimensional) temporal structure $(\Omega, \mathcal{V})$ where Ω is the set of all time periods and a valuation function $\mathcal{V} : P \to 2^\Omega$ from a set P of propositions to their valuation sets. We say a temporal structure is homogeneous if all propositions are homogeneous as we have in this paper. Then the point-free semantics of MTL is defined as follows.

Definition 4.1 (Point-free Semantics of MTL). The point-free satisfaction relation $\vDash$ of a past metric temporal logic formula φ in a homogeneous temporal structure $(\Omega, \mathcal{V})$, relative to a time period $(t,t') \in \Omega$ is defined as follows:

$$\begin{array}{lcl}
(t,t') \vDash p & \leftrightarrow & (t,t') \in \mathcal{V}(p) \\
(t,t') \vDash \neg\varphi & \leftrightarrow & \forall t < r < r' < t.\ (r,r') \nvDash \varphi \\
(t,t') \vDash \varphi_1 \wedge \varphi_2 & \leftrightarrow & (t,t') \vDash \varphi_1 \text{ and } (t,t') \vDash \varphi_2 \\
(t,t') \vDash \varphi_1\ \mathcal{S}_{(a,b)}\ \varphi_2 & \leftrightarrow & \forall t < t'' < t'.\ \exists r < r' < t''. \\
 & & \quad (r,r') \vDash \varphi_2 \text{ and} \\
 & & \quad (r',t'') \vDash \varphi_1 \text{ and} \\
 & & \quad a < t'' - r' < b
\end{array}$$

It is clear how to extend these definitions to the other Boolean operators $\vee$ and $\rightarrow$ using their definition in terms of $\wedge$ and $\neg$ as well as to other past temporal operators $\blacklozenge_{(a,b)}$ and $\blacksquare_{(a,b)}$ using their definition in terms of $\mathcal{S}_{(a,b)}$. Using these definitions, we extend the valuation function $\mathcal{V}$ from propositions to past MTL formulas such that $\mathcal{V}(\varphi) = \{(t,t') \mid (t,t') \vDash \varphi\}$. Then we have that the homogeneity property is preserved for valuation sets of formulas.

Proposition 4.2. *For any past MTL formula φ and point-free dense-time behavior w, the valuation set $\mathcal{V}(\varphi)$ is homogeneous.*

We now explain our point-free semantics with examples. For example, consider the disjunction of two behaviors $p = \{(3,4)\}$ and $q = \{(4,6)\}$. We first calculate the negated sets $\mathcal{V}(\neg p) = \{(r,r') \mid 0 \le r < r' \le 3 \text{ or } 4 \le r < r'\}$ and $\mathcal{V}(\neg q) = \{(r,r') \mid 0 \le r < r' \le 4 \text{ or } 6 \le r < r'\}$ according to the point-free semantics. Then we apply the set intersection to get $\mathcal{V}(\neg p \wedge \neg q) = \{(r,r') \mid 0 \le r < r' \le 3 \text{ or } 6 \le r < r'\}$ and finally obtain $\mathcal{V}(p \vee q) = \{(r,r') \mid 3 \le r < r' \le 6\}$ by negating again, which is equivalent to the desired behavior $p \vee q = \{(3,6)\}$. Note that this calculation is for illustration purposes as we already mentioned more efficient algorithms to concretely perform Boolean operations over point-free behaviors. The point-free version of the since operator holds on a time period (t,t') if there exists a time period ending at r' that satisfies φ_2 and a time period that satisfies φ_1 on (r',t'') for all $t < t'' < t'$. By the homogeneity property, φ_1 holds for all

time periods (r', r'') for $r' < r'' < t''$; therefore, the semantics fulfills the intuitive meaning of the Since operation.

In the following we study the relation between the traditional pointy semantics and our point-free semantics. The foremost connection is that an MTL formula φ holds on a time period (t, t') if φ holds for all time points in (t, t') expressed as

$$\forall t < t'' < t'.\ t'' \vdash \varphi \ \rightarrow\ (t, t') \vDash \varphi \qquad \text{(CONT)}$$

Observe that all four types of continuous intervals, namely $[t, t']$, $[t, t')$, $(t, t']$, and (t, t'), are mapped to the same time period (t, t'). Then we can reduce any pointy dense time behavior into a point-free one using this mapping. Clearly the implication CONT does not hold in the other direction as the point-free semantics ignores (the value of singular) points, in particular, boundary points and removable discontinuities.

It is observed that the pointy model of time becomes more well-behaving under finite variability and left continuity restrictions. The finite variability is a common restriction that limits the discussion to finitely representable behaviors. The left continuity restriction over Boolean behaviors is captured by the identity

$$t' \vdash \varphi \leftrightarrow \exists t < t'.\ \forall t < t'' < t'.\ t'' \vdash \varphi \qquad \text{(LC)}$$

that says that, for every time point t', there exist a time point $t < t'$ in the past and the value of all points in between are equal to the value of t'. Recall that the original assumption of the pointy model is that every point is an isolated (individual) member of the timeline. However, by assuming left continuity, we would actually contradict with the pointy foundations and accept that (the satisfaction of) a time point is not that isolated but depends on its (immediate) past. Therefore, we see this restriction applied more in engineering and physics (with its classical definition with limits) than computer science and mathematical literature.

The following result shows that this practice (restricting the point model under variability and continuity) makes the pointy model essentially point-free.

THEOREM 4.3. *The pointy semantics ($\vdash_{FV,LC}$) under finite variability (FV) and left continuity (LC) conditions is equivalent to the flattening of the point-free semantics ($\vDash$) onto the second (end) component such that*

$$t' \vdash_{FV,LC} \varphi \quad \leftrightarrow \quad \exists t < t'.\ (t, t') \vDash \varphi$$

PROOF. By induction on the formula structure and directly for each case. See Appendix C. □

In other words, we can obtain homogeneous point-free valuations from a sequence of left-open and right closed intervals (as left continuity enforces) and reconstruct them by projecting homogeneous valuations onto the second axis.

Finally note that these results in this section can be easily extended to include future temporal logic by using symmetric definitions (the until instead of the since, right-continuity instead of left-continuity, etc.) if desired.

5 IMPLEMENTATION AND EVALUATION

We have implemented our approach using code generation technique similar to other works [11, 12]. For a given past MTL formula, the monitor construction algorithm traverses the syntax tree of the formula and generates corresponding state variables, updates equations, and the output function in our target programming language, C++, for each subformula. Since these are valid C++ statements, we encapsulate them in an ordinary C++ class having two methods, namely update and output. The generated class corresponds to our sequential network based monitor and it is the final output of our implementation. Note that we employ Interval Container Library (ICL) distributed as a part of Boost C++ Libraries for operations on sets of time periods in updates equations and output functions. The next thing is to write a main application code that instantiates the generated (monitor) class and repeatedly call its update and output methods while reading the input. This part is currently not automated but we provide an example main application file to read Comma Separated Values (CSV) files, feed the monitor incrementally, and print the output. The monitor class and the rest of application are then compiled using the standard `g++` compiler with `-O2` optimizations. In the following, we perform all tests on a laptop with a 2.50GHz Intel Core i5-7200 processor and 8GB memory.

We first compare our discrete-time monitors, generated by our implementation called REELAY, with two publicly available monitoring tools, namely MONPOLY and AERIAL, that support online past MTL monitoring. Both tools use an event (sample) based time model, which is equivalent to discrete model if we have an event (sample) for each discrete point (See [5] for the details and comparison). We check in our tests whether tools scale for large timing constraints since we usually have to reason about slow and fast changing processes together and the monitoring process should not be affected by changing the base time unit (e.g. from seconds to milliseconds). We start our comparison by a property

$$\text{QPR}(a, b) = \blacksquare\big((r \wedge \neg q \wedge \blacklozenge q) \rightarrow (p\ \mathcal{S}_{[a,b]}\ q)\big)$$

that intuitively says the proposition p always holds between (the last occurrence of) q and r for a duration in $[a, b]$. We depict our comparison results in Table 7 where we see that MONPOLY and REELAY can handle large timing constraints very well whereas AERIAL does not scale. In this case, we are about 15 times faster than MONPOLY but such difference is in the margin of the implementation details including the choice of programming language (Ocaml for MONPOLY) and our explicit compilation strategy.

The second property $\text{PANDQ}(a, b) = p\ \mathcal{S}_{[a,b]}\ q$ checks a specific condition such that the subformula q occurs frequently (continuously at the extreme) and the upper bound b is large. For example, we encounter such a scenario when a safety controller monitors a system under oversampling and timing constraints are usually much larger than the sampling period. We are then testing tools over an extreme input where φ_1 and φ_2 hold continuously. From Table 7, we see that this case breaks MONPOLY's algorithm as it explicitly stores all occurrences of φ_2 in the past time window bounded by b. On the other hand, our monitor perform well and require a constant time as the time bound b is getting larger.

Next we try to break our monitors as well. As mentioned in Section 3, we have a very specific worst-case condition that involves (1) a formula $p\ \mathcal{S}_{[a,b]}\ q$ with a very narrow and large timing constraint such that $b - a \ll b$, and (2) an input behavior of q with frequent short pulses. This scenario is captured by our third property $\text{DELAY}(b) = p\ \mathcal{S}_{[a,b]}\ q$ over p holds continuously and q holds every other time point. It is seen from Table 7 the monitoring

	Aerial	MonPoly	Reelay
qpr(3,6)	1.613	1.857	**0.121**
qpr(30,60)	5.761	1.734	**0.106**
qpr(300,600)	69.29	1.734	**0.102**
pandq(1,6)	1.353	2.725	**0.363**
pandq(1,60)	5.553	7.426	**0.364**
pandq(1,600)	65.72	61.59	**0.364**
delay(6)	1.397	2.256	**0.347**
delay(60)	5.488	4.186	**1.553**
delay(600)	63.665	25.972	**13.176**

Table 7: Comparing the performance of tools for three temporal properties with varying timing parameters over discrete time behaviors of length 1 million.

time is linearly increasing with respect to b since we need to store $b/2$ number of (disjoint) periods in the set V_k. For other cases, in particular for formulas with wider timing bounds, the algorithm mostly needs to store large overlapping time periods so we are able to merge them and keep the effective size of the set small. Note that this scenario is also hard for Aerial and MonPoly.

Finally we test our dense time monitors in comparison with the discrete as there is no publicly available tool for dense/continuous semantics to our knowledge (despite some algorithms exist [5, 15]). Recall that we usually observe continuous processes via fixed-step oversampling, which may lead to long discrete behaviors with long stutters. Using discrete monitors is of course a source of inefficiency in this case and dense time monitors can process such chunks of stuttering periods at one step. In our experiments, we cap the maximum length Δ of chunks. This means that a step of the dense monitor would change between 1 and Δ time units depending on the length of stuttering periods (whereas it is always 1 for the discrete). Other advantages of dense monitors include that they can work with rational (or floating) numbers and variable-step (asynchronous) sampling schemes.

There is an overhead of dense monitors (due to heavier Boolean operations and synchronization of behaviors) but we illustrate in Table 8 that it pays off while we are processing long stuttering behaviors. For example, we see the dense monitor of pandq(1,600) is two times slower than the discrete one with $\Delta = 10$. However, it processes 22% faster than the discrete for $\Delta = 100$, and 43% faster for $\Delta = 1000$. We also report that two monitors have almost the same speed when the cap is 35 for this particular example. It suggests that dense monitors can become more effective starting from stuttering lengths as low as 35 —an oversampling ratio of 256 is common in digital signal processing. The other two properties, pandq(1,600) and delay(600), are outliers as the input behavior for the former always stutters and for the latter never does. Then we first see the speed of monitoring for pandq(1,600) is approaching to the speed of reading the CSV file. Second, the overhead of dense monitors is visible in the case of delay(600) as there is no stutters a dense monitor can exploit. Therefore, we conclude that discrete monitors should be preferred for rapidly changing processes and dense time monitors for slowly changing processes.

Max. Length of Chunk (Δ)	10	100	1000
qpr(300,600)	0.50x	**1.22x**	**1.43x**
pandq(1,600)	**2.39x**	**5.55x**	**6.50x**
delay(600)	0.18x	0.18x	0.18x

Table 8: Speed of dense monitors relative to discrete monitors under different maximum chunk sizes.

6 RELATED WORK

The pioneering work [12] for monitoring proposes a simple technique to construct online monitors from past LTL specifications. This construction has been extended for timed and quantitative (data) properties in several subsequent works [4, 5, 11, 20]. In terms of tools and implementation, this branch (network-based monitors as we called) is quite fruitful perhaps because of its simplicity. In Section 5, we compare our timed extension with some of these tools [3, 6]. Other approaches for online MTL monitoring include automata based [13, 16], tester based [17], and the incremental marking procedures [15]. To our knowledge, no implementations are available for these automata and tester based monitors. The implementation of the incremental algorithm has been reported to work only with an external simulator in [15] and is not available in the current version [18].

From a semantic point of view, dense time monitoring algorithms are divided into sample or event based [4, 13, 20] and interval based continuous [15–17] semantics. An extensive comparison of two sorts can be found in [5] with a conclusion that interval-based semantics are more natural and sample-based are more efficient. In this paper, we propose a point-free semantics for MTL as the third sort with a claim that it is both natural and efficient. Finally, there are other formalisms that can be regarded as point-free. These are timed regular expressions under so-called signal semantics [2] and interval temporal logics under strict semantics [10, 22], which excludes punctual intervals. We use similar definitions and techniques with these works in our formalization and a brief review of interval logics is included in Appendix A.

7 CONCLUSIONS

We described how to construct online monitors based on sequential networks from past metric temporal logic (MTL) specifications over discrete and dense time behaviors. Our proposed construction and monitors are simple, compositional, extensible, and easily implementable. These properties are especially important when online monitors are deployed in real-time systems and used actively in a larger (control) system. In our experiments, we demonstrated that our monitors can handle large time constraints over long sequences more efficiently than similar existing tools. Contrary to the commonly-held belief, our dense time monitors are no more complicated than the discrete ones. We showed that dense time monitors perform even better for so-called stuttering behaviors, which are usually obtained in monitoring continuous processes and systems via sampling.

The theoretical contribution of this paper is the introduction of a point-free semantics for MTL by discarding time points from the

formulation. In that we were mainly concerned with the physical grounding and practicality, which the traditional pointy time model fails to provide. Therefore, our focus in the paper was in showing the (mathematical and practical) benefits of having a point-free model for dense time monitoring solutions. We can easily say that, together with sequential networks, the simplicity of our dense time monitors is due to such point-free model of time. We believe that the point-free approach can be also beneficial in other research areas in timed systems including verification and synthesis.

REFERENCES

[1] James F Allen. 1983. Maintaining Knowledge about Temporal Intervals. *Commun. ACM* 26, 11 (1983), 832–843.

[2] Eugene Asarin, Paul Caspi, and Oded Maler. 2002. Timed Regular Expressions. *J. ACM* 49, 2 (2002), 172–206.

[3] David Basin, Matúš Harvan, Felix Klaedtke, and Eugen Zălinescu. 2011. MONPOLY: Monitoring usage-control policies. In *International Conference on Runtime Verification (RV)*. Springer, 360–364.

[4] David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. 2015. Monitoring Metric First-Order Temporal Properties. *J. ACM* 62, 2 (2015), 15.

[5] David Basin, Felix Klaedtke, and Eugen Zălinescu. 2018. Algorithms for monitoring real-time properties. *Acta Informatica* 55, 4 (2018), 309–338.

[6] David A Basin, Srdjan Krstic, and Dmitriy Traytel. 2017. AERIAL: Almost Event-Rate Independent Algorithms for Monitoring Metric Regular Properties.. In *RV-CuBES*. 29–36.

[7] Janusz A. Brzozowski and Ernst Leiss. 1980. On Equations for Regular Languages, Finite Automata, and Sequential Networks. *Theoretical Computer Science* 10, 1 (1980), 19–35.

[8] Irving M Copi, Calvin C Elgot, and Jesse B Wright. 1958. Realization of Events by Logical Nets. *J. ACM* 5, 2 (1958), 181–196.

[9] Bradley Dowden. [n. d.]. Zeno's Paradoxes. In *The Internet Encyclopedia of Philosophy, ISSN 2161-0002*. Accessed: October 24, 2018.

[10] Joseph Y Halpern and Yoav Shoham. 1991. A Propositional Modal Logic of Time Intervals. *J. ACM* 38, 4 (1991), 935–962.

[11] Klaus Havelund, Doron Peled, and Dogan Ulus. 2017. First-Order Temporal Logic Monitoring with BDDs. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD)*.

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

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

[14] Ron Koymans. 1990. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems* 2, 4 (1990), 255–299.

[15] Oded Maler and Dejan Ničković. 2013. Monitoring Properties of Analog and Mixed-Signal Circuits. *International Journal on Software Tools for Technology Transfer* 15, 3 (2013), 247–268.

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

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

[18] Dejan Nickovic, Olivier Lebeltel, Oded Maler, Thomas Ferrère, and Dogan Ulus. 2018. 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)*.

[19] Amir Pnueli. 1977. The Temporal Logic of Programs. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*. 46–57.

[20] Prasanna Thati and Grigore Roşu. 2005. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science* 113 (2005), 145–162.

[21] Dogan Ulus, Thomas Ferrère, Eugene Asarin, and Oded Maler. 2014. Timed Pattern Matching. In *Proceedings of the Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*. 222–236.

[22] Dogan Ulus and Oded Maler. 2018. Specifying Timed Patterns using Temporal Logic. In *Proceedings of the Conference on Hybrid Systems: Computation and Control (HSCC)*.

[23] Yde Venema. 1990. Expressiveness and Completeness of an Interval Tense Logic. *Notre Dame Journal of Formal Logic* 31, 4 (1990), 529–547.

A TEMPORAL LOGICS OF TIME PERIODS

In this section we give the overview of temporal logics based on time periods (rather than time points) and period-based temporal modalities used in these logics. We then reveal their connection with our point-free semantics and, in particular, we give equivalent definitions of the point-free MTL operators in terms of period-based temporal modalities.

Considering time periods as primitive entities, Allen introduced thirteen basic relations between two time periods to represent high-level temporal knowledge in [1]. The set of so-called Allen's relations between time periods consists of relations *met-by* (A), *begins* (B), *ends* (E), *during* (D), *overlaps* (O), and *later* (L) as well as their inverses and the equality ($=$). Halpern and Shoham applied modal logic approach over time periods in [10] and proposed a temporal logic (HS-logic) that features a temporal modality for each relation above. It is shown that six certain temporal modalities of the HS-logic can express others under strict semantics, which excludes points. Other properties of HS-logic can be found in [23]. The metric extension of HS logic, called Metric Compass Logic (MCL), is introduced by Ulus and Maler in [22] by the following grammar:

$$\varphi \;:=\; p \mid \overline{\varphi} \mid \varphi_1 \cap \varphi_2 \mid \langle X \rangle_{(a,b)}\, \varphi$$

where $p \in P$ is a proposition, $X \in \{A, \overline{A}, B, \overline{B}, E, \overline{E}\}$ is a relation between time periods, and (a, b) is a timing constraint for the modality. The existential temporal modality $\langle X \rangle_{(a,b)}\, \varphi$ is the dual of an universal temporal modality $[X]_{(a,b)}\, \varphi = \overline{\langle X \rangle_{(a,b)}\, \overline{\varphi}}$ as usual. The satisfaction relation $\models$ of an MCL formula φ in a temporal structure $(\Omega, \mathcal{V})$, relative to a time period $(t, t') \in \Omega$ is defined as follows:

$$
\begin{array}{lll}
(t,t') \models p & \leftrightarrow & (t,t') \in \mathcal{V}(p) \\
(t,t') \models \overline{\varphi} & \leftrightarrow & (t,t') \not\models \varphi \\
(t,t') \models \varphi_1 \cap \varphi_2 & \leftrightarrow & (t,t') \models \varphi_1 \text{ and } (t,t') \models \varphi_2 \\
\\
(t,t') \models \langle B \rangle_{(a,b)}\, \varphi & \leftrightarrow & \exists t < t'' < t'.\ (t,t'') \models \varphi \\
 & & \quad \text{and } a < t' - t'' < b \\
(t,t') \models \langle \overline{B} \rangle_{(a,b)}\, \varphi & \leftrightarrow & \exists t'' > t'.\ (t,t'') \models \varphi \\
 & & \quad \text{and } a < t'' - t' < b \\
(t,t') \models \langle E \rangle_{(a,b)}\, \varphi & \leftrightarrow & \exists t < t'' < t'.\ (t'',t') \models \varphi \\
 & & \quad \text{and } a < t'' - t < b \\
(t,t') \models \langle \overline{E} \rangle_{(a,b)}\, \varphi & \leftrightarrow & \exists t'' < t.\ (t'',t') \models \varphi \\
 & & \quad \text{and } a < t - t'' < b \\
(t,t') \models \langle A \rangle_{(a,b)}\, \varphi & \leftrightarrow & \exists t'' > t'.\ (t',t'') \models \varphi \\
 & & \quad \text{and } a < t'' - t' < b \\
(t,t') \models \langle \overline{A} \rangle_{(a,b)}\, \varphi & \leftrightarrow & \exists t'' < t.\ (t'',t) \models \varphi \\
 & & \quad \text{and } a < t - t'' < b
\end{array}
$$

The following defines point-free MTL operators introduced in this paper using MCL operators given in this section. Given the formulas φ_1 and φ_2 satisfy the homogeneity, it is easy to check the equivalences

$$
\begin{aligned}
\neg\varphi_1 &\equiv [B][E]\,\overline{\varphi_1} \equiv [E][B]\,\overline{\varphi_1} \\
\varphi_1 \wedge \varphi_2 &\equiv \varphi_1 \cap \varphi_2 \\
\varphi_1\, \mathcal{S}_{(a,b)}\, \varphi_2 &\equiv [B]\langle E \cup \overline{E} \rangle \left(\varphi_1 \cap \langle \overline{A} \rangle_{(a,b)}\, \varphi_2 \right)
\end{aligned}
$$

where $\langle E \cup \overline{E} \rangle\, \varphi \equiv \langle E \rangle\, \varphi \cup \langle \overline{E} \rangle\, \varphi$.

B GENERATING A MONITOR

In this section, we give more details about our monitor generation procedure from past MTL specifications. Monitor generation starts with a specification file that contains an MTL property and other related information. For example, in Figure 3, we present the specification file `qpr10.yaml` for the property QPR(3, 6) used in our experiments. It contains data fields such as `pattern` that specifies the past temporal logic formula and `time_model` that is either discrete or dense.

```
---
name : "QPR10"
lang : "temporal"
time_model : "discrete"
pattern : "(r && !q && once q ) -> (p since[3,6] q)"
with_headers : true
```

Figure 3: The specification file `qpr10.yaml`

We then execute our implementation REELAY to generate the monitor class as given in Figure 4 and required headers such as `discrete_time.hpp`, which contains discrete time update equations presented in this paper.

```
#include "array"
#include "discrete_time.hpp"

template<typename T = int>
struct MonitorQPR10 {

    std::array<bool,5> states = std::array<bool
     ,5>{0,0,0,0,0};
    std::array<bool,5> states_pre = std::array<bool
     ,5>{0,0,0,0,0};

    std::array<interval_set<T>,1> tstates = std::array<
     interval_set<T>,1>{interval_set<T>()};
    std::array<interval_set<T>,1> tstates_pre = std::
     array<interval_set<T>,1>{interval_set<T>()};

    T now = 0;

    bool update(bool p, bool q, bool r){

        now = now + 1;
        states_pre = states;
        tstates_pre = tstates;

        states[0] = not(q);
        states[1] = (r and states[0]);
        states[2] = (states_pre[2] or q);
        states[3] = (states[1] and states[2]);
        tstates[0] = update_timed_since<T>(tstates_pre
     [0], now, p, q, 3, 6);
        states[4] = (not(states[3]) or output_timed_since
     <T>(tstates[0], now));

        return output();
    }

    bool output(){
        return states[4];
    }
};
```

Figure 4: The monitor class file `MonitorQPR10.hpp`

```
#include "iostream"

#include "array"
#include "csv.h"

#include "MonitorQPR10.hpp"

int main(int argc, char **argv){

    int q, p, r;
    bool output;

    if (argc < 1) {
        std::cout <<  "ERROR: No input file" << std::endl
     ;
        return 0;
    }

    auto monitor = MonitorQPR10<int>();

    io::CSVReader<3> reader(argv[1]);
    reader.read_header(io::ignore_extra_column, "q", "p",
      "r");

    while(reader.read_row(q, p, r)){
        monitor.update(p, q, r);
        output = monitor.output();
        if(not output){
            std::cout << "Violation at time " << monitor.
     now << std::endl;
        }
    }

    return 0;
}
```

Figure 5: The main application code

Our implementation employs Interval Container Library (ICL) distributed with Boost C++ libraries to perform set operations over sets of time periods. Techniques shown in the paper allow us to use such existing libraries directly, which reduces the burden of implementation greatly and improves the overall quality.

Figure 5 shows the main application code, which is hand written, where we instantiate the generated monitor for a specific purpose. For our experiments, we read a Comma Separated Value (CSV) file using an off-the-shelf CSV reader implementation contained in `csv.h` and print an error message if a violation is detected. Since we cannot know all the use cases beforehand, we prefer generating a self-contained class file from formal specifications and not restricting the user in other design decisions such as file formats. We believe that this is a good strategy for both parties.

Finally, we obtain the actual monitoring program by compiling these source files together using standard tools and run it over an input file such as the one presented in Figure 6.

```
q,p,r
1,0,0
0,1,0
0,1,0
0,1,0
0,1,0
0,1,1
```

Figure 6: A discrete time input behavior file in CSV format

C PROOF OF THEOREM 4.3

Provided the left continuity condition (LC) such that

$$t' \vdash \varphi \leftrightarrow \exists t < t'.\ \forall t < t'' < t'.\ t'' \vdash \varphi$$

and finite variability (FV), the proof first proceeds by the induction of the formula structure. The base case of propositions and Boolean operations are straightforward. We prove

$$t' \vdash_{\text{FV,LC}} \ \varphi_1 \ \mathcal{S} \ \varphi_2 \ \ \leftrightarrow \ \ \exists t < t'.\ (t, t') \vDash \varphi_1 \ \mathcal{S} \ \varphi_2$$

For the direction ($\rightarrow$), assume that the formula $\varphi_1 \ \mathcal{S} \ \varphi_2$ holds at a time point t', then we directly have the following steps:

(1) $\exists t < t'.\ \forall t < t'' < t'.\ t'' \vdash \varphi_1 \ \mathcal{S} \ \varphi_2$ (LC)
(2) $\exists t < t'.\ \forall t' < t'' < t'.$ ($\mathcal{S}$)
$\exists r' < t''.\ r' \vdash \varphi_2$ and $\forall r' < r'' < t''.\ r'' \vdash \varphi_1$
(3) $\exists t < t'.\ \forall t' < t'' < t'.$ ($\vDash$)
$\exists r' < t''.\ \exists r < r'.\ (r, r') \vDash \varphi_2$ and $(r', t'') \vDash \varphi_1$
(4) $\exists t < t'.\ (t, t') \vDash \varphi_1 \ \mathcal{S} \ \varphi_2$ ($\mathcal{S}$)

For the other direction ($\leftarrow$), assume that the formula $\varphi_1 \ \mathcal{S} \ \varphi_2$ holds on a a time period (t, t'), then we directly have the following steps:

(1) $\exists t < t'.\ \forall t < t'' < t'.\ \exists r < r' < t''.$ ($\mathcal{S}$)
$(r, r') \vDash \varphi_2$ and $(r', t'') \vDash \varphi_1$
(2) $\exists t < t'.\ \forall t < t'' < t'.\ \exists r < r' < t''.$ (HOM)
$(r, r') \vDash \varphi_2$ and
$\forall r' < r'' < r''' < t''.\ (r'', r''') \vDash \varphi_1$
(3) $\exists t < t'.\ \forall t < t'' < t'.\ \exists r' < t''.$
$\exists r < r'.\ (r, r') \vDash \varphi_2$ and
$\forall r' < r''' < t''.\ \exists r''.\ (r'', r''') \vDash \varphi_1$
(4) $\exists t < t'.\ \forall t < t'' < t'.\ t'' \vdash \varphi_1 \ \mathcal{S} \ \varphi_2$ ($\mathcal{S}$)
(5) $t' \vdash \varphi_1 \ \mathcal{S} \ \varphi_2$ (LC)

Hence, we obtain ($\leftrightarrow$). □