Verified Online Monitoring for Metric Temporal Logic with Lattice-based Quantitative Semantics

Agnishom Chattopadhyay(✉) and Konstantinos Mamouras

Rice University, Houston, USA
{agnishom, mamouras}@rice.edu

Abstract. We investigate the formalization, using the Coq proof assistant, of a procedure for constructing online monitors from specifications written in past-time metric temporal logic (MTL). We employ an algebraic quantitative semantics that encompasses the Boolean and robustness semantics of MTL and we interpret formulas over a discrete temporal domain. A potentially infinite-state variant of Mealy machines, a kind of string transducers, is used as a formal model of online monitors. The main result is that there is a compositional construction from formulas to monitors, so that each monitor computes (in an online fashion) the semantic values of the corresponding formula over the input stream. From our Coq formalization, we extract OCaml code for executable online monitors. We have compared the performance of our monitoring framework with Reelay, a state-of-the-art tool for monitoring temporal properties.

Keywords: Online Monitoring · Formal Verification · Quantitative Semantics.

1 Introduction

Verifying cyber-physical systems statically is usually infeasible at large scales, and may require making assumptions about the behavior of the environment. In contrast, runtime verification is a lightweight technique for checking that a system exhibits the desired behavior. It is often performed in an *online* fashion, which means that the execution trace of the system is observed as it is being generated. This trace typically consists of one or more signals and event streams. A *monitor* program runs in parallel with the system, consumes the system trace incrementally, and outputs at every step a value that summarizes the current state of the system. This value can be a Boolean indication of whether an interesting event or pattern has been identified, or it can contain richer quantitative information. There is a substantial amount of existing work on formalisms for specifying monitors, as well as on algorithms for their efficient execution.

Temporal patterns are often specified using logical formalisms. Linear Temporal Logic (LTL) is one such widely utilized formalism which admits efficient algorithms. It is common to constrain the occurrence of temporal patterns using time intervals, which is a feature that gives rise to an extension of LTL called

metric temporal logic (MTL) [40]. Since many applications in the domain of cyber-physical systems frequently deal with comparison between numerical signals, Signal Temporal Logic (STL) [42], an extension of LTL with predicates allowing comparison with numerical values, is widely used.

While temporal logic facilitates the specification of temporal properties, it is equally important to have accompanying algorithms. The notion of a monitor is an algorithm which analyzes given traces for a specific temporal property. In an offline setting, the trace is available in its entirety. In contrast, online monitors are meant to be attached to running systems, so that they may report interesting (or critical) events as they happen, potentially so that a supervisor can act in real time. Thus, they must analyze system traces incrementally (fragment by fragment) as they evolve and this must be done efficiently: each update should be handled quickly.

The standard semantics for temporal logic is qualitative, which means that monitors classify traces only in a binary pass/fail manner. However, this is not sufficiently informative for certain applications: some violations can be more serious than others, and on the other hand, some cases of satisfaction could be close to the edge of failure. In some cases, we may be able to take corrective action if we could tell that the system is approaching a potential violation. Indeed, in realistic systems with continuous dynamics, some degree of tolerance must be allowed since every value is accurate only up to the extent of measurement errors. This encourages us to consider quantitative semantics for our formalisms, so that we can quantify how robustly the observed behavior fits the desired specification [31].

The variant of MTL that we consider in this paper is interpreted over a discrete temporal domain and is a past-time only fragment of the logic. In the setting of online monitoring we need to reactively respond to the patterns in what we have seen so far. So, using a past-time fragment makes sense and provides a clean semantics. Online monitoring with future-time temporal connectives has been considered, but these can give rise to semantic complications [29].

Using the interactive theorem prover Coq [56], we formalize the semantics of our temporal logic. The implementation of our monitoring algorithms are done within Coq, and a proof of correctness is given. Formal proofs, like the ones described in Coq, are thoroughly rigorous and machine checkable. This gives us confidence in the correctness of our implementations. With the extraction mechanism of Coq, we can obtain executable OCaml code directly from our verified implementation.

It would be difficult to deploy an OCaml-based implementation on an embedded device in a cyber-physical system due to scarcity of runtime resources. However, our verified monitor could be used as a part of the development-level environment for such systems. An example of such an activity is the use of runtime monitoring for the purpose of falsification (see [30]). Our verified online monitor could also be used in any scenario where offline monitoring can be used. Another utility of our monitor is that it could be used as an oracle for differen-

tially testing the correctness of other monitors. We demonstrate this possibility in our paper by finding some bugs in the Reelay monitoring tool.

As mentioned earlier, a strong motivation for using a quantitative semantics is to quantify how robustly a signal satisfies a given specification in view of potential perturbations. One way to do so for STL specifications is to interpret formulas over real numbers and interpret the logical connectives $\vee$ and $\wedge$ as max and min respectively [27]. In our work, we use a slightly more general framework, interpreting our formulas over arbitrary bounded distributive lattices. This abstract algebraic framework enables a simpler verification approach and, as we will discuss later, does not hurt the performance of our algorithms.

In our formalization, we model online monitors as a potentially infinite-state variant of Mealy machines. They are abstract machines whose state evolves as fragments of a trace are consumed. Each state of the machine is associated with a value that represents the current output of the monitor. We follow a compositional approach for our implementation and proofs. This is done with the help of combinators, which are constructs that compose Mealy machines in different ways (possibly with other data structures) so that their behaviors can be composed or combined. Corresponding to each Boolean or temporal connective in our specification language, we identify a combinator on Mealy machines which implements the desired behavior.

We observe that formulas in our temporal logic can be rewritten so that only a few combinators are necessary: (1) combinators which combine the output of Mealy machines running in parallel by applying a binary operation on their respective outputs, (2) combinators which compute a running aggregate on the results of a Mealy machine, (3) combinators which compute running aggregates over sliding windows, and (4) combinators which withhold the results of a machine until a given number of updates. We will see that most of these can be implemented in a straightforward way. Applying a binary operation to the current output values of two running machines can be done with a stateless construction. Computing running aggregates efficiently can be achieved by storing the aggregate of the trace seen so far. In order to withhold the results of a given machine, we can simply store them in a queue of a fixed length. Computing aggregates over sliding windows is slightly trickier. This is usually achieved with an algorithm that maintains monotonic wedges [41]. However, this assumes that the semantic values are totally ordered, which is not necessarily true in our setting of lattices. Instead, we use an algorithm that is inspired from the well-known implementation of a queue data structure using two stacks, popular in functional programming. A variant of this algorithm can be used for computing sliding-window aggregates for any associative operation in a way that every execution step of the monitor needs $O(1)$ amortized time.

Our Coq formalization and extracted code are available in a public GitHub Repository[1].

[1] `https://github.com/Agnishom/lattice-mtl`

Outline of the paper. In Sect. 2, we first introduce lattices and then present the syntax and semantics of our temporal specification language. In Sect. 3, we give a formal definition of Mealy machines, present a collection of Mealy combinators, and discuss in detail their implementation. In Sect. 4, we discuss the extraction of executable OCaml code from the Coq scripts, use it as a verification oracle and we compare its performance against the monitoring tool Reelay [59]. Finally, in Sect. 5, we discuss several different quantitative semantics for Signal Temporal Logic, various algorithmic approaches to online monitoring, and we also give a brief overview of related efforts to produce formally verified monitors.

2 Metric Temporal Logic

In this section, we review metric temporal logic (MTL), which will be the formalism that we consider here for specifying quantitative properties. We use bounded distributive lattices as semantic value domains for our logic. While this abstract algebraic setting is not usually how MTL is interpreted, we will see that the standard qualitative (Boolean) and quantitative (robustness) semantics can be obtained simply by choosing the appropriate lattice.

2.1 Lattices

A lattice is a partial order in which every two elements have a least upper bound and a greatest lower bound. We will use an equivalent algebraic definition.

Definition 1. A *lattice* is a set A together with associative and commutative binary operations $\sqcap$ and $\sqcup$, called *meet* and *join* respectively, that satisfy the *absorption laws*, i.e, $x \sqcup (x \sqcap y) = x$ and $x \sqcap (x \sqcup y) = x$ for all $x, y \in A$.

Let A be a lattice. Using the absorption laws it can be shown that $\sqcup$ is idempotent: $x \sqcup x = x \sqcup (x \sqcap (x \sqcup x)) = x$ for every $x \in A$. Similarly, it can also be shown that $\sqcap$ is idempotent. Define the relation $\sqsubseteq$ as follows: $x \sqsubseteq y$ iff $x \sqcup y = y$ for all $x, y \in A$. The relation $\sqsubseteq$ is a partial order. It also holds that $x \sqsubseteq y$ iff $x \sqcap y = x$. For all $x, y \in A$, the element $x \sqcup y$ is the supremum (least upper bound) of $\{x, y\}$ and the element $x \sqcap y$ is the infimum (greatest lower bound) of $\{x, y\}$ w.r.t. the order $\sqsubseteq$.

Definition 2. A lattice A is said to be *bounded* if there exists a *top* element $\top \in A$ and a *bottom* element $\bot \in A$ such that $\bot \sqcup x = x$ and $x \sqcap \top = x$ (equivalently, $\bot \sqsubseteq x \sqsubseteq \top$) for every $x \in A$.

Let A be a bounded lattice. It is easy to check that $x \sqcup \top = \top$ and $\bot \sqcap x = \bot$ for every $x \in A$. For a finite subset $X = \{x_1, x_2, \ldots x_n\}$ of a bounded lattice, we write $\bigsqcup X$ for $x_1 \sqcup x_2 \sqcup \cdots \sqcup x_n$ and similarly $\bigsqcap X$ for $x_1 \sqcap x_2 \sqcap \cdots \sqcap x_n$. Moreover, we define $\bigsqcup \emptyset$ to be $\bot$ and $\bigsqcap \emptyset$ to be $\top$. So, $\bigsqcup X$ is the supremum of X and $\bigsqcap X$ is the infimum of X.

Definition 3. A lattice A is said to be *distributive* if $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$ and $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$ for all $x, y, z \in A$.

Example 4. Consider the two-element set $\mathbb{B} = \{\top, \bot\}$ of Boolean values, where $\top$ represents truth and $\bot$ represents falsity. The set $\mathbb{B}$, together with conjunction as meet and disjunction as join, is a bounded and distributive lattice.

Example 5. The set $\mathbb{R}$ of real numbers, together with min as meet and max as join, is a distributive lattice. However, $(\mathbb{R}, \min, \max)$ is not a bounded lattice. It is commonplace to adjoin the elements ∞ and $-\infty$ to $\mathbb{R}$ so that they serve as the top and bottom element respectively.

2.2 Syntax and Semantics

We fix a set $\mathbb{D}$ of *data items*. We denote by $\mathbb{D}^\omega$ the set of infinite sequences over $\mathbb{D}$, which can also be thought of as functions of type $\mathbb{N} \to \mathbb{D}$. We call members of $\mathbb{D}^\omega$ *traces*. We also consider non-empty strings over $\mathbb{D}$, denoted $\mathbb{D}^+$, which we call (trace-)*prefixes*. Given a trace σ, we use $\sigma|_n$ to denote the finite string $\sigma(0)\sigma(1)\cdots\sigma(n)$.

We also fix a bounded distributive lattice $\mathbb{V}$, whose elements are *quantitative truth values* that represent degrees of truth or falsity. Given a formula, our quantitative semantics will associate a truth value (from $\mathbb{V}$) with each position of the trace. The set Φ of temporal formulas that we consider is given by the following grammar:

$$\varphi, \psi ::= f : \mathbb{D} \to \mathbb{V} \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \mathsf{P}_I \varphi \mid \mathsf{H}_I \varphi \mid \varphi \, \mathsf{S}_I \, \psi \mid \varphi \, \overline{\mathsf{S}}_I \, \psi,$$

where I is an interval of the form $[a, b]$ or $[a, \infty)$ with $a, b \in \mathbb{N}$. For every temporal connective $X \in \{\mathsf{P}, \mathsf{H}, \mathsf{S}, \overline{\mathsf{S}}\}$, we will write X_a as an abbreviation for $X_{[a,a]}$ and X as an abbreviation for $X_{[0,\infty)}$. We interpret formulas from Φ over traces $\mathbb{D}^\omega$ at specific positions using the *robustness* interpretation function $\rho : \Phi \times \mathbb{D}^\omega \times \mathbb{N} \to \mathbb{V}$, defined as follows:

$$\begin{aligned}
\rho(f, \sigma, i) &= f(\sigma(i)) \\
\rho(\varphi \vee \psi, w, i) &= \rho(\varphi, \sigma, i) \sqcup \rho(\psi, \sigma, i) \\
\rho(\varphi \wedge \psi, \sigma, i) &= \rho(\varphi, \sigma, i) \sqcap \rho(\psi, \sigma, i) \\
\rho(\mathsf{P}_I \varphi, \sigma, i) &= \bigsqcup_{\substack{j \in I \\ i-j \geq 0}} \rho(\varphi, \sigma, i-j) \\
\rho(\mathsf{H}_I \varphi, \sigma, i) &= \bigsqcap_{\substack{j \in I \\ i-j \geq 0}} \rho(\varphi, \sigma, i-j) \\
\rho(\varphi \, \mathsf{S}_I \, \psi, \sigma, i) &= \bigsqcup_{\substack{j \in I \\ i-j \geq 0}} \Big(\rho(\psi, \sigma, i-j) \sqcap \bigsqcap_{k<j} \rho(\varphi, \sigma, j-k) \Big) \\
\rho(\varphi \, \overline{\mathsf{S}}_I \, \psi, \sigma, i) &= \bigsqcap_{\substack{j \in I \\ i-j \geq 0}} \Big(\rho(\psi, \sigma, i-j) \sqcup \bigsqcup_{k<j} \rho(\varphi, \sigma, j-k) \Big)
\end{aligned}$$

Note that $\rho(\mathsf{P}_a\varphi, \sigma, i) = \bot$ and $\rho(\mathsf{H}_a\varphi, \sigma, i) = \top$ whenever $a > i$. The semantics of $\wedge$, H and $\overline{\mathsf{S}}$ can be obtained from that of $\vee$, P and S by switching the roles of $\sqcap$ and $\sqcup$. Thus, they will be referred to as dual operators.

Since we are interpreting formulas over discrete traces, our logic is expressively equivalent to LTL with a "Previous" operator. In other words, temporal connectives (including S and $\overline{\mathsf{S}}$; see Lemma 14) with bounded intervals can be rewritten in terms of multiple compositions of the Previous operator instead. Also, note that our temporal logic does not include negation. However, this does not limit expressiveness as we discuss in the examples below.

Example 6. Continuing from Example 4, we choose $\mathbb{D}$ to be $\mathbb{B}^k$ and we set $\mathbb{V}$ to $\mathbb{B}$. The set of functions from $\mathbb{B}^k \to \mathbb{B}$ considered may be restricted to projections $\pi_i(b_1, \ldots b_i, \ldots b_k) = b_i$ and negated projections $\pi_i(b_1, \ldots b_i, \ldots b_k) = \overline{b_i}$. This gives us the standard qualitative semantics for metric temporal logic. Formulas with negation can be expressed equivalently as formulas in negation normal form (NNF) in a fairly standard way by pushing negation inside while interchanging operators for their dual operators.

Example 7. We can also express a past-time version of STL interpreted over discrete time in this framework. To do so, take $\mathbb{D} = \mathbb{R}^k$. A qualitative semantics is obtained by taking $\mathbb{V}$ to be $\mathbb{B}$ and restricting the functions to comparisons of the form $(r_1, \ldots, r_i, \ldots, r_k) \mapsto r_i \sim c$ where $c \in \mathbb{R}$ and $\sim \in \{\leq, \geq, =\}$. A quantitative semantics can be obtained by taking $\mathbb{V}$ to be $\mathbb{R} \cup \{\infty, -\infty\}$ (as in Example 5) and considering functions of the form $(r_1, \ldots, r_i, \ldots, r_k) \mapsto r_i - c$ or $(r_1, \ldots r_i \ldots r_k) \mapsto c - r_i$. Even in the quantitative setting, STL formulas with negation can be presented in our framework by considering NNF, again by pushing negation inside while interchanging operators for their dual operators and replacing $r_i - c$ with $c - r_i$.

Our formalism is a *past-time only* logic. This means that the robustness value at a point can be determined by the trace prefix up to that position. This idea can be stated formally in the form of the following claim.

Lemma 8. Suppose σ, τ are traces such that $\sigma|_n = \tau|_n$ for some $n \in \mathbb{N}$. Then, for any formula $\varphi \in \Phi$ and for every $i \leq n$, $\rho(\varphi, \sigma, i) = \rho(\varphi, \tau, i)$.

This suggests a way to interpret formulas on trace-prefixes. Suppose $w \in \mathbb{D}^+$, and $\sigma \in \mathbb{D}^\omega$ is some trace such that $\sigma|_{|w|} = w$. Then, we define $\rho(\varphi, w) = \rho(\varphi, \sigma, |w|)$. Lemma 8 implies that this definition does not depend on the specific choice of σ.

3 The Monitoring Problem

Monitoring is the processing of an input trace in order to detect specified patterns. For quantitative properties, this could be thought of as applying a valuation function on a trace. In an online setting, the trace is supplied to the monitor incrementally. To elaborate, the monitor consumes fragments of the trace one

at a time and the monitor is required to evaluate the quantitative property on the trace prefix seen so far. Below, we outline a compositional approach for monitoring quantitative properties denoted by MTL formulas.

3.1 Monitors as Mealy Machines

We will use a variant of Mealy machines, a class of string transducers, as a formal model of online monitoring algorithms.

Definition 9. Let A and B be sets. A *Mealy machine* with input items from A and output values in B is a tuple $(\mathtt{St}, \mathtt{init}, \mathtt{mNext}, \mathtt{mOut})$ where $\mathtt{St}$ is a (possibly infinite) set of states, $\mathtt{init} \in \mathtt{St}$ is the initial state, $\mathtt{mNext} : \mathtt{St} \times A \to \mathtt{St}$ is a transition function which transitions the state of the machine upon seeing an input from A, and $\mathtt{mOut} : \mathtt{St} \times A \to B$ provides an output at the current state, given an element from A. We write $\mathtt{Mealy}(A, B)$ for the set of all Mealy machines with inputs from A and outputs from B.

While this is similar to the standard definition of Mealy Machines found in the literature, we use an equivalent, co-inductive definition in our formalization. In the co-inductive view (see [21]), the states are not explicitly expressed, but described directly in terms of their extensional behavior.

```
CoInductive Mealy (A B : Type) : Type := {
  mOut : A -> B;
  mNext : A -> Mealy A B;
}.
```

The functions `mNext` and `mOut` denote the incremental update and output of the machine, respectively, which consume traces element by element. We can extend these functions to `gNext` and `gOut` to consume non-empty strings, more generally. We can think of the function `gOut` as the quantitative property that the machine associates with the given trace-prefix.

Definition 10. Let $m \in \mathtt{Mealy}(A, B)$. Then, $\mathtt{gNext}(m) : A^* \to \mathtt{Mealy}(A, B)$ is defined by $\mathtt{gNext}(m, \varepsilon) = m$ and $\mathtt{gNext}(m, w \cdot a) = \mathtt{mNext}(\mathtt{gNext}(m, w), a)$. We define $\mathtt{gOut}(m) : A^+ \to B$ by $\mathtt{gOut}(m, w \cdot a) = \mathtt{mOut}(\mathtt{gNext}(m, w), a)$.

For a quantitative property of trace-prefixes, i.e, a function $f : \mathbb{D}^+ \to \mathbb{V}$, we wish to construct a Mealy machine that computes f. In particular, we are interested in quantitative properties which arise as denotations of MTL formulas.

Definition 11. Let $\varphi \in \Phi$ and $m \in \mathtt{Mealy}(\mathbb{D}, \mathbb{V})$. We say that the Mealy machine m *implements a monitor for* φ if $\mathtt{gOut}(m, w) = \rho(\varphi, w)$ for all $w \in \mathbb{D}^+$.

Example 12. Following Definition 9, consider the machine $m : \mathtt{Mealy}(\mathbb{V}, \mathbb{V})$ with states $\mathbb{V}$ (indicating that it stores one element of type $\mathbb{V}$), initial state $\bot$, $\mathtt{mOut}(u, a) = u$ and $\mathtt{mNext}(u, a) = a$. It holds that $\mathtt{gOut}(m, v_1) = \bot$ and $\mathtt{gOut}(m, v_1 v_2) = v_1$, $\mathtt{gOut}(m, v_1 v_2 v_3) = v_2$, etc. The machine m implements a monitor for the formula $\mathsf{P}_1(v \mapsto v)$ in the sense of Definition 11.

Stated formally, the monitoring problem for MTL is to find a translation $\texttt{toMonitor} : \Phi \to \texttt{Mealy}(\mathbb{D}, \mathbb{V})$ so that given any $\varphi \in \Phi$, $\texttt{toMonitor}(\varphi)$ implements a monitor for φ.

3.2 Monitor Combinators

Combinators are compositional constructs that let one define new machines in terms of existing ones. Our approach towards solving the monitoring problem is to find combinators which correspond to the temporal and Boolean connectives of MTL. With these combinators, a monitor for a given formula can be specified by induction on the structure of the formula. A similar approach for MTL with bounded future-time connectives is considered in [50,46]. The compositional construction of transducers, called temporal testers, for temporal formulas has been studied in [53,44,34]. The use of combinators for specifying more general computations for stream processing has been considered in the design of the domain-specific languages StreamQRE [48] and StreamQL [39]. Quantitative regular expressions (QREs) [7,48] (see also [8] and [11]) are particularly relevant. QREs have been used to specify complex algorithms for medical monitoring [1,3]. Moreover, the relationship between QREs and automata-theoretic models with registers is investigated in [9,10,6].

Proceeding with the idea of compositional monitor construction, we identify the key constructs which are necessary in achieving the expressive power of MTL. We say that the formulas φ and ψ are *equivalent*, and we write $\varphi \equiv \psi$, if $\rho(\varphi, \sigma, i) = \rho(\psi, \sigma, i)$ for all traces $\sigma \in \mathbb{D}^\omega$ and positions $i \in \mathbb{N}$.

Lemma 13. The following identities hold:

$$\mathsf{P}_{[a,b]}\varphi \equiv \mathsf{P}_a \mathsf{P}_{[0,b-a]}\varphi \tag{1}$$

$$\mathsf{H}_{[a,b]}\varphi \equiv \mathsf{H}_a \mathsf{H}_{[0,b-a]}\varphi \tag{2}$$

$$\varphi\, \mathsf{S}_{[a+1,b]}\, \psi \equiv \mathsf{H}_{[0,a]}\varphi \wedge \mathsf{P}_{a+1}\left(\varphi\, \mathsf{S}_{[0,b-(a+1)]}\, \psi\right) \tag{3}$$

$$\varphi\, \overline{\mathsf{S}}_{[a+1,b]}\, \psi \equiv \mathsf{P}_{[0,a]}\varphi \vee \mathsf{H}_{a+1}\left(\varphi\, \overline{\mathsf{S}}_{[0,b-(a+1)]}\, \psi\right) \tag{4}$$

$$\mathsf{P}_{[a,\infty)}\varphi \equiv \mathsf{P}_a \mathsf{P}_{[0,\infty)}\varphi \tag{5}$$

$$\mathsf{H}_{[a,\infty)}\varphi \equiv \mathsf{H}_a \mathsf{H}_{[0,\infty)}\varphi \tag{6}$$

$$\varphi\, \mathsf{S}_{[a+1,\infty)}\, \psi \equiv \mathsf{H}_{[0,a]}\varphi \wedge \mathsf{P}_{a+1}\left(\varphi\, \mathsf{S}_{[0,\infty)}\, \psi\right) \tag{7}$$

$$\varphi\, \overline{\mathsf{S}}_{[a+1,\infty)}\, \psi \equiv \mathsf{P}_{[0,a]}\varphi \vee \mathsf{H}_{a+1}\left(\varphi\, \overline{\mathsf{S}}_{[0,\infty)}\, \psi\right) \tag{8}$$

The proofs of the identities of Lemma 13 are straightforward. Proving the identities involving S (or $\overline{\mathsf{S}}$) requires the distributivity axioms, which motivates the need for considering distributive lattices.

Lemma 14. The following identities hold:

$$\varphi\, \mathsf{S}_{[0,a]}\, \psi \equiv (\varphi\, \mathsf{S}\, \psi) \wedge \mathsf{P}_{[0,a]}\psi \tag{9}$$

$$\varphi\, \overline{\mathsf{S}}_{[0,a]}\, \psi \equiv (\varphi\, \overline{\mathsf{S}}\, \psi) \vee \mathsf{H}_{[0,a]}\psi \tag{10}$$

Proof. We will only prove the first identity, since the second one can be proved by dualizing the same argument. Let $\sigma \in \mathbb{D}^\omega$ be an arbitrary trace and $n \in \mathbb{N}$ a position. We define $s_i = \rho(\varphi, \sigma, n-i)$ and $t_i = \rho(\psi, \sigma, n-i)$ for every $i \in \mathbb{N}$. Then, we have that

$$\begin{aligned}
\rho(\varphi \,\mathsf{S}_{[0,a]}\, \psi, \sigma, n) &= \bigsqcup_{i \le K} \Big(t_i \sqcap \bigsqcap_{j<i} s_j \Big) \\
\rho(\varphi \,\mathsf{S}\, \psi, \sigma, n) &= \bigsqcup_{i \le n} \Big(t_i \sqcap \bigsqcap_{j<i} s_j \Big) = \rho(\varphi \,\mathsf{S}_{[0,a]}\, \psi, \sigma, n) \sqcup \bigsqcup_{K < i \le n} \Big(t_i \sqcap \bigsqcap_{j<i} s_j \Big) \\
\rho(\mathsf{P}_{[0,a]} \psi, \sigma, n) &= \bigsqcup_{i \le K} t_i
\end{aligned}$$

where $K = \min(a, n)$. We have to prove that $L = R \sqcap Q$, where $L = \rho(\varphi \,\mathsf{S}_{[0,a]}\, \psi, \sigma, n)$, $R = \rho(\varphi \,\mathsf{S}\, \psi, \sigma, n)$ and $Q = \rho(\mathsf{P}_{[0,a]} \psi, \sigma, n)$. From $K \le n$ we obtain that $L \sqsubseteq R$. It also holds that $L \sqsubseteq Q$ because $t_i \sqcap \bigsqcap_{j<i} s_j \sqsubseteq t_i$ for every $i \le K$. It follows that $L \sqsubseteq R \sqcap Q$. It remains to show that $R \sqcap Q \sqsubseteq L$. Since

$$\begin{aligned}
R \sqcap Q &= \Big(L \sqcup \bigsqcup_{K < i \le n} \Big(t_i \sqcap \bigsqcap_{j<i} s_j \Big) \Big) \sqcap Q \\
&= (L \sqcap Q) \sqcup \bigsqcup_{K < i \le n} \Big(t_i \sqcap \bigsqcap_{j<i} s_j \sqcap Q \Big) \\
&= (L \sqcap Q) \sqcup \bigsqcup_{K < i \le n} \bigsqcup_{k \le K} \Big(t_i \sqcap t_k \sqcap \bigsqcap_{j<i} s_j \Big),
\end{aligned}$$

it suffices to establish that $L \sqcap Q \sqsubseteq L$ (which is true) and $t_i \sqcap t_k \sqcap \bigsqcap_{j<i} s_j \sqsubseteq L$ for every i and k with $K < i \le n$ and $k \le K$. Since $k < i$, we conclude that $t_i \sqcap t_k \sqcap \bigsqcap_{j<i} s_j \sqsubseteq t_k \sqcap \bigsqcap_{j<k} s_j \sqsubseteq L$. ◻

Remark 15. In the qualitative setting, the identities of Lemma 14 are intuitively clear, but they require a more careful argument in the quantitative setting. They have been used and proven in [26] for the lattice $(\mathbb{R}, \min, \max)$, but the given proof does not generalize to the class of lattices that we consider here. As we can see in the proof of Lemma 14, there is a subtlety in dealing with the terms of $\rho(\varphi \,\mathsf{S}\, \psi, \sigma, n)$ with index $i = K+1, \ldots, n$.

The first set of identities allows us to express $\mathsf{P}_{[\bullet,\bullet]}$, $\mathsf{S}_{[\bullet,\bullet]}$ in terms of $\mathsf{P}_{[0,\bullet]}$, $\mathsf{S}_{[0,\bullet]}$ and $\mathsf{P}_\bullet$. The second set of identities implies that $\mathsf{S}_{[0,\bullet]}$ can be replaced by S and $\mathsf{P}_\bullet$. Thus, the only additional constructs required in expressing the bounded temporal operators are $\mathsf{P}_\bullet$ and $\mathsf{P}_{[0,\bullet]}$ (and their duals).

We present in Figure 1 a summary of the combinators that we will consider. Each combinator can be thought of as the implementation of the corresponding Boolean or temporal connective. The key observation is that this association between combinators on Mealy machines and connectives *respect* the implementation relation (Definition 11) between machines and formulas. E.g., if m is a monitor for φ, we expect `mSometimeBounded` k m to be a monitor for $\mathsf{P}_{[0,k]}\varphi$.

$$\frac{f : \mathbb{D} \to \mathbb{V}}{\texttt{mAtomic}\ f : \texttt{Mealy}(\mathbb{D}, \mathbb{V})} \qquad \frac{m : \texttt{Mealy}(\mathbb{D}, \mathbb{V}) \qquad k : \mathbb{N}}{\texttt{mDelay}\ k\ m : \texttt{Mealy}(\mathbb{D}, \mathbb{V})} \qquad \frac{m : \texttt{Mealy}(\mathbb{D}, \mathbb{V}) \qquad k : \mathbb{N}}{\overline{\texttt{mDelay}}\ k\ m : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}$$

$$\frac{m_1 : \texttt{Mealy}(\mathbb{D}, \mathbb{V}) \qquad m_2 : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}{\texttt{mAnd}\ m_1\ m_2 : \texttt{Mealy}(\mathbb{D}, \mathbb{V})} \qquad \frac{m_1 : \texttt{Mealy}(\mathbb{D}, \mathbb{V}) \qquad m_2 : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}{\texttt{mOr}\ m_1\ m_2 : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}$$

$$\frac{m_1 : \texttt{Mealy}(\mathbb{D}, \mathbb{V}) \qquad m_2 : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}{\texttt{mSince}\ m_1\ m_2 : \texttt{Mealy}(\mathbb{D}, \mathbb{V})} \qquad \frac{m_1 : \texttt{Mealy}(\mathbb{D}, \mathbb{V}) \qquad m_2 : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}{\overline{\texttt{mSince}}\ m_1\ m_2 : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}$$

$$\frac{m : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}{\texttt{mSometime}\ m : \texttt{Mealy}(\mathbb{D}, \mathbb{V})} \qquad \frac{m : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}{\texttt{mAlways}\ m : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}$$

$$\frac{m : \texttt{Mealy}(\mathbb{D}, \mathbb{V}) \qquad k : \mathbb{N}}{\texttt{mSometimeBounded}\ k\ m : \texttt{Mealy}(\mathbb{D}, \mathbb{V})} \qquad \frac{m : \texttt{Mealy}(\mathbb{D}, \mathbb{V}) \qquad k : \mathbb{N}}{\texttt{mAlwaysBounded}\ k\ m : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}$$

Fig. 1. Summary of Mealy Combinators

$$[f] = \texttt{mAtomic}\ f$$

$$\begin{aligned}
[\varphi \wedge \psi] &= \texttt{mAnd}\ [\varphi]\ [\psi] & [\varphi \vee \psi] &= \texttt{mOr}\ [\varphi]\ [\psi] \\
[\mathsf{P}\varphi] &= \texttt{mSometime}\ [\varphi] & [\mathsf{H}\varphi] &= \texttt{mAlways}\ [\varphi] \\
[\mathsf{P}_{[0,a]}\varphi] &= \texttt{mSometimeBounded}\ a\ [\varphi] & [\mathsf{H}_{[0,a]}\varphi] &= \texttt{mAlwaysBounded}\ a\ [\varphi] \\
[\mathsf{P}_{a}\varphi] &= \texttt{mDelay}\ a\ [\varphi] & [\mathsf{H}_{a}\varphi] &= \overline{\texttt{mDelay}}\ a\ [\varphi] \\
[\varphi\ \mathsf{S}\ \psi] &= \texttt{mSince}\ [\varphi]\ [\psi] & [\varphi\ \overline{\mathsf{S}}\ \psi] &= \overline{\texttt{mSince}}\ [\varphi]\ [\psi] \\
[\mathsf{P}_{[a,\infty)}\varphi] &= [\mathsf{P}_{a}\mathsf{P}_{[0,\infty)}\varphi] \quad (a > 0) & [\mathsf{H}_{[a,\infty)}\varphi] &= [\mathsf{H}_{a}\mathsf{H}_{[0,\infty)}\varphi] \quad (a > 0) \\
[\mathsf{P}_{[a,b]}\varphi] &= [\mathsf{P}_{a}\mathsf{P}_{[0,b-a]}\varphi] \quad (a > 0, a \neq b) & [\mathsf{H}_{[a,b]}\varphi] &= [\mathsf{H}_{a}\mathsf{H}_{[0,b-a]}\varphi] \quad (a > 0, a \neq b) \\
[\varphi\ \mathsf{S}_{[a+1,b]}\ \psi] &= [\mathsf{H}_{[0,a]}\varphi \wedge \mathsf{P}_{a+1}(\varphi\ \mathsf{S}_{[0,b-(a+1)]}\ \psi)] & [\varphi\ \overline{\mathsf{S}}_{[a+1,b]}\ \psi] &= [\mathsf{P}_{[0,a]}\varphi \vee \mathsf{H}_{a+1}(\varphi\ \overline{\mathsf{S}}_{[0,b-(a+1)]}\ \psi)] \\
[\varphi\ \mathsf{S}_{[a+1,\infty)}\ \psi] &= [\mathsf{H}_{[0,a]}\varphi \wedge \mathsf{P}_{a+1}(\varphi\ \mathsf{S}\ \psi)] & [\varphi\ \overline{\mathsf{S}}_{[a+1,\infty)}\ \psi] &= [\mathsf{P}_{[0,a]}\varphi \vee \mathsf{H}_{a+1}(\varphi\ \overline{\mathsf{S}}\ \psi)] \\
[\varphi\ \mathsf{S}_{[0,a]}\ \psi] &= [(\varphi\ \mathsf{S}\ \psi) \wedge \mathsf{P}_{[0,a]}\psi] & [\varphi\ \overline{\mathsf{S}}_{[0,a]}\ \psi] &= [(\varphi\ \overline{\mathsf{S}}\ \psi) \vee \mathsf{H}_{[0,a]}\psi]
\end{aligned}$$

Fig. 2. The `toMonitor` function

The definition of the `toMonitor` function which constructs monitors from formulas is shown in Figure 2. As discussed, it proceeds by rewriting the formula into the desired form and then replacing each temporal or Boolean connective with the corresponding combinator. The main correctness claim for the monitor is stated as follows:

```
Theorem toMonitor_correctness:
  forall φ, implements (toMonitor φ) φ.
```

The proof of this theorem is done using induction on the structure of the formula. Once the identities in Lemmas 13 and 14 are proven, this can be done using the correctness properties of individual combinators.

Before we start describing each combinator in detail, we make some remarks about the general organization of our implementation and formal proofs. There is a lot of symmetry among these combinators that can be leveraged for economy of effort. An example is the presence of dual connectives. This is why in

```
CoFixpoint mAtomic {A B : Type} (f : A -> B) : Mealy A B :=
  {| mOut x := f x;
     mNext _ := mAtomic f; |}.
Lemma mAtomic_state {A B : Type} (f : A -> B) (l : nonEmpty A) :
  gNext (mAtomic f) l = mAtomic f.
Lemma mAtomic_result {A B : Type} (f : A -> B) (l : nonEmpty A) :
  gOut (mAtomic f) l = f (latest l).
Lemma monAtomic_correctness :
  forall f, implements (monAtomic f) (FAtomic f).
```

Fig. 3. Establishing correctness of `mAtomic`.

many cases we focus on presenting these combinators in a slightly general way before instantiating them specifically to $\texttt{Mealy}(\mathbb{D}, \mathbb{V})$. As discussed before, the correctness for each combinator is phrased in terms of preserving the implementation relation – these theorems are indexed with the suffix `correctness`. These theorems are proven via lemmas indexed with the suffix `result` which characterize the most recent output of the Mealy machine at some point in the computation. The proofs proceed by induction on the trace seen so far. They require additional lemmas that establish invariants about the state of a Mealy machine as it evolves during the computation. These latter lemmas are indicated with the suffix `state`. These ideas are illustrated in the construction of `mAtomic` in Figure 3.

Atomic Functions. In order to lift functions $f : A \to \mathbb{V}$ to $\texttt{Mealy}(A, \mathbb{V})$, we define the `mAtomic` combinator, as shown in Figure 3. Given a function $f : A \to \mathbb{V}$, it defines a Mealy machine which applies f to the latest input element. We use the lemma `mAtomic_state` to describe the evolution of the machine when an arbitrary stream prefix is fed. Using this, we also prove `mAtomic_result`, which describes the final output of the machine after accepting an arbitrary stream prefix. The lemma titled `mAtomic_correctness` establishes that `mAtomic` correctly translates atomic functions to corresponding monitors.

Pointwise Binary Operations. In Figure 4, we define the combinator `mBinOp` that combines the output of two given machines using a binary operation. By plugging in $\sqcup$ and $\sqcap$ as `op`, we can use `mBinOp` to implement the $\vee$ and $\wedge$ connectives, respectively. Like in the case of `mAtomic`, the correctness of this combinator is proven by establishing appropriate lemmas which describe the behavior of `mBinOp` with `gNext` and `gOut`. These let us prove, in particular, that `mAnd` and `mOr` correctly implement formulas involving $\wedge$ and $\vee$, respectively.

Delay Monitors. We view the implementation of $\mathsf{P}_\bullet$ and $\mathsf{H}_\bullet$ as a mechanism that delays the output of a Mealy Machine. For instance, the sequence

```
CoFixpoint mBinOp {A B C D : Type} (op : B -> C -> D)
 (m : Mealy A B) (n : Mealy A C) : Mealy A D := {|
  mOut (a : A) := op (mOut m a) (mOut n a);
  mNext (a : A) := mBinOp op (mNext m a) (mNext n a);
|}.
Definition monAnd (m n : Mealy A Val) : Mealy A Val :=
  mBinOp meet m n.
Lemma monAnd_correctness m1 m2 φ1 φ2 :
  implements m1 φ1 -> implements m2 φ2
  -> implements (monAnd m1 m2) (FAnd φ1 φ2).
Definition monOr (m n : Mealy A Val) : Mealy A Val :=
  mBinOp join m n.
Lemma monOr_correctness m1 m2 φ1 φ2 :
  implements m1 φ1 -> implements m2 φ2
  -> implements (monOr m1 m2) (FOr φ1 φ2).
```

Fig. 4. The `mBinOp` combinator

```
Lemma delayWith_state (q : Queue) (m : Mealy A B) (l : nonEmpty A) :
    forall initSeg, initSeg = (back q) ++ rev (front q)
 -> forall k, k = length initSeg
 -> forall stream, stream = (toList (gCollect m l)) ++ initSeg
 -> forall lastSeg, lastSeg = firstn k stream
 -> exists newFront newBack,
               lastSeg = newBack ++ rev newFront
             /\ length lastSeg = k
             /\ gNext (delayWith q m) l
                = delayWith (Build_Queue newFront newBack) (gNext m l).
```

Fig. 5. Delay monitors.

$\langle \rho(\mathsf{P}_2\varphi, a_1a_2a_3), \rho(\mathsf{P}_2\varphi, a_1a_2a_3a_4)\rangle$ is same as $\langle \rho(\varphi, a_1), \rho(\varphi, a_1a_2)\rangle$. These operators preserve the order of the outputs, but delay them by a given constant.

This can be achieved using a queue maintained at a fixed length. For instance, to implement $\mathsf{P}_a\varphi$, we maintain a queue of length a. Upon being given an input item $a \in \mathbb{D}$, we feed a to $\texttt{toMonitor}(\varphi)$, enqueue the result and then return what we obtain by dequeuing. This works since the dequeued element was the result of $\texttt{toMonitor}(\varphi)$ a turns ago. The queue needs to be initially filled with a instances of $\bot$ (or $\top$ in the case of H_a) since we have that $\rho(\mathsf{P}_a\varphi, w) = \bot$ (or $\rho(\mathsf{H}_a\varphi, w) = \top$) when $|w| > a$.

Since Coq is based on a functional programming environment, functional lists are the ordered collections that are the easiest for us to reason about and work with. Functional lists are typically implemented via linked lists, which means that in order to access the kth element of the list, one would have to traverse k links and would spend $O(k)$ time. This makes appending to the end of the list expensive. However, obtaining or adding elements at the head (the

```
CoFixpoint mFold {A B : Type} (op : B -> B -> B)
 (m : Mealy A B) (init : B) : Mealy A B := {|
  mOut (a : A) := op init (mOut m a);
  mNext (a : A) := mFold op (mNext m a) (op init (mOut m a)); |}.
Definition mSometime (m : Mealy A Val) : Mealy A Val :=
  mFold join m bottom.
Definition monAlways (m : Mealy A Val) : Mealy A Val :=
  mFold meet m top.
```

Fig. 6. Temporal Folds

beginning) of the list is straightforward. Thus, these lists effectively behave as stacks and sometimes we refer to them as such. We use the well-known technique of implementing a queue with two functional lists, which we briefly discuss below.

A queue is represented by two lists `front` and `rear`. When an element is enqueued, it is added to the head of the `rear` list. Thus, the `rear` list effectively stores the elements of the queue in an order opposite to that in which they were enqueued. When dequeing an element is required, the elements of `rear` are reversed and placed in the `front` (thus restoring the order) and the head of `front` is returned. As long as `front` is non-empty, subsequent dequeues may be directly handled by returning the head of `front`.

In our use case, the queue is maintained at a fixed length, say k, and every enqueue is followed by a subsequent dequeue. Reversing `rear` into `front` takes time $O(k)$. However, we only need to do this every k turns, since `front` is filled with k items whenever the reversal happens. Thus, every k turns, we do $O(k)$ work and only $O(1)$ work is needed otherwise. This gives us an amortized time complexity of $O(1)$.

We implement this idea in the `delayWith` combinator in Figure 5. The key lemma required in proving the correctness of the `delayWith` combinator shows that the queue maintained always stores the last k-many outputs of the sub-monitor. To formalize this, we define $\texttt{gCollect} : \texttt{Mealy}(A, B) \times \mathbb{D}^+ \to \mathbb{V}^+$ as

$$\begin{aligned}&\texttt{gCollect}(m, a_1 a_2 \cdots a_n) = \\ &\langle \texttt{gOut}(m, a_1), \texttt{gOut}(m, a_2), \cdots, \texttt{gOut}(m, a_1 a_2 \cdots a_n)\rangle.\end{aligned}$$

We may now write the mentioned invariant as in `delayWith_state`, which is established by induction on the input stream.

Temporal Folds. The unbounded operators P and H can be thought of as a running fold on the input stream, since $\rho(\mathsf{P}\varphi, w \cdot a) = \rho(\mathsf{P}\varphi, w) \sqcup \rho(\varphi, w \cdot a)$ (and similarly for H). Thus, to evaluate these operators in an online fashion, we only need to store the robustness value for the trace seen so far. For P (resp., H), the robustness of the current trace can then be obtained by computing the join (resp., meet) of the current value and the stored one. In Figure 6, `mAlways` (resp., `mSometime`) computes the robustness values corresponding to the H (resp., P)

```
CoFixpoint sinceAux (m1 m2 : Mealy A Val) (pre : Val) : Mealy A Val :=
  {| mOut (a : A) :=  (mOut m2 a) ⊔ (pre ⊓ (mOut m1 a));
     mNext (a : A) :=
      sinceAux (mNext m1 a) (mNext m2 a)
        ((mOut m2 a) ⊔ (pre ⊓ (mOut m1 a)))
  |}.

Definition monSince (m1 m2 : Mealy A Val) : Mealy A Val :=
  sinceAux m1 m2 bottom.
```

Fig. 7. Monitoring Since

connectives by computing the meet (resp., join) of the current value with the stored one. Their proof of correctness is a straightforward induction on the trace-prefix, using the incremental equation involving the operator.

Using the following identity, we may also view the computation of S as a temporal fold, i.e, the robustness for $\varphi \,\mathsf{S}\, \psi$ may be calculated incrementally by only storing the robustness value for the stream prefix so far.

Lemma 16. For all $w \in \mathbb{D}^+$ and $a \in \mathbb{D}$, we have that

$$\rho(\varphi \,\mathsf{S}\, \psi, w \cdot a) = \rho(\psi, w \cdot a) \sqcup (\rho(\varphi \,\mathsf{S}\, \psi, w) \sqcap \rho(\varphi, w \cdot a)).$$

This is a well known equality and can be proved by using distributivity in a straightforward way. A proof of this for the $(\mathbb{R}, \max, \min)$ lattice appears in [24].

Using the equality of Lemma 16, `mSince` can be implemented as in Figure 7. The correctness of `mSince` is established by proving invariants on `mSinceAux`, which is straightforward once the equality above has been established.

Windowed Temporal Folds. For the operators $\mathsf{P}_{[0,a]}$ or $\mathsf{H}_{[0,a]}$, the strategy above needs to be modified, since the fold is over a sliding window, rather than the entire trace. For this purpose, we use a queue like data structure (dubbed `aggQueue`, henceforth) which also maintains sliding window aggregates, in addition. An extended discussion of a similar data structure can be found in [18].

While we intend to use `aggQueue` specifically for computing sliding window join and meet on bounded lattices, the algorirthmic idea behind the data structure only uses two ideas involving $\sqcup$ (resp., $\sqcap$). Namely: (1) $\sqcup$ (resp., $\sqcap$) is associative (2) $\bot$ (resp., $\top$) are identities for $\sqcup$ (resp., $\sqcap$). These features make the lattice elements a monoid under $\sqcup$ (resp., $\sqcap$). in the remainder of this section, we will describe the algorithm for a monoid $(\mathtt{B}, \cdot, \mathbb{1})$.

As the name suggests, we can think of `aggQueue` as a data structure with a queue-like interface: it supports operations $\mathtt{aggEnqueue} : \mathtt{aggQueue} \times \mathtt{B} \to \mathtt{aggQueue}$ and $\mathtt{aggDequeue} : \mathtt{aggQueue} \to \mathtt{aggQueue}$ which allow enqueuing elements of `B` or dequeueing them. However, instead of a peek operation, we are interested in an aggregate operation $\mathtt{agg} : \mathtt{aggQueue} \to \mathtt{B}$ which reports the aggregate of all the elements in the queue.

```
Definition aggQueue_inv (l : list B) (q : aggQueue) :=
    exists olds news,
      olds ++ news = l
      /\ new q = rev news
      /\ newAgg q = finite_op _ (rev (new q))
      /\ length (oldAggs q) = length olds
      /\ forall i , nth i (oldAggs q) unit = finite_op _ (skipn i olds).

Definition agg_inv (l : list B) (q : aggQueue) :=
    agg q = finite_op _ l.
Lemma aggQueue_agg_inv l q :
    aggQueue_inv l q -> agg_inv l q.

Lemma enqueue_aggQueue_inv l q :
    aggQueue_inv l q
    -> forall n, aggQueue_inv (l ++ [n]) (aggEnqueue n q).

Lemma aggDequeue_aggQueue_inv x xs q :
    aggQueue_inv (x :: xs) q
    -> aggQueue_inv xs (aggDequeue q).
```

Fig. 8. Invariants for `aggQueue`

To implement an usual queue, we maintained two lists: a rear list into which the `new` elements are enqueued, and a front list from which elements can be dequeued. Here, since we are interested in only knowing the aggregates, we replace the front list with an `oldAggs` list, which stores partial aggregates instead. Additionally, we keep track of `newAgg`, the aggregate of the values in `new`. Suppose that the contents of the represented queue are a list `l`. Then, the invariant we want to maintain suggests that `l` can be broken into two parts `olds` and `news` such that (1) `new` contains the elements of `news` in reverse order (2) `newAgg` is the aggregate of the elements of `news` (3) `oldAggs` has the same length as that of `olds` (4) The i-th element of `oldAggs` is the aggregate of the $|\texttt{olds}| - i$ elements of `olds`. Given these invariants, it is easy to see that the aggregate of the entire queue can be computed as the aggregate of `newAgg` and the head of `oldAggs`.

We maintain these invariants in the following way: Upon enqueuing $b \in \texttt{B}$, we simply add b to the head of `new` and update `newAgg` to $\texttt{newAgg} \cdot b$. Performing a dequeue is easy when `oldAggs` is non-empty: we simply remove the element at its head. When `oldAggs` is empty, the contents of `new` are added as incremental partial aggregates to `oldAggs`. In Figure 8, we show a formalization of the invariants that one needs to prove.

To keep a sliding window aggregate of the last k elements, `mSometimeBounded` (or `mAlwaysBounded`) initializes an `aggQueue` filled with k instances of $\mathbb{1}$ (i.e, $\bot$ (or $\top$)). When a new input is available, the monitor enqueues the result of the corresponding submonitor into queue and dequeues the element which was enqueued k turns ago. The output of the machine is simply the aggregate of

`l`	`new`	`oldAggs`	`newAgg`	`agg`
$\langle \mathbb{1}, \mathbb{1}, \mathbb{1} \vert \rangle$	$\langle\rangle$	$\langle \mathbb{1}, \mathbb{1}, \mathbb{1} \rangle$	$\mathbb{1}$	$\mathbb{1}$
$\langle \mathbb{1}, \mathbb{1} \vert a \rangle$	$\langle a \rangle$	$\langle \mathbb{1}, \mathbb{1} \rangle$	a	$\mathbb{1} \cdot a$
$\langle \mathbb{1} \vert a, b \rangle$	$\langle b, a \rangle$	$\langle \mathbb{1} \rangle$	ab	$\mathbb{1} \cdot ab$
$\langle \vert a, b, c \rangle$	$\langle c, b, a \rangle$	$\langle\rangle$	abc	$\mathbb{1} \cdot abc$
$\langle b, c \vert d \rangle$	$\langle d \rangle$	$\langle bc, c \rangle$	d	$bc \cdot d$
$\langle c \vert d, e \rangle$	$\langle e, d \rangle$	$\langle c \rangle$	de	$c \cdot de$

Fig. 9. A run of the sliding window algorithm that aggregates the last 3 elements. The elements a, b, c, d, e are fed in, incrementally. We use | as a seperator in `l` to indicate the old and the new parts of the queue. Note that the contents of `l` itself is not stored.

the elements in the queue. Using a similar argument as before, we can see that the invocations of `mNext` on these machines run in $O(1)$ amortized time (with a worst case behaviour of $O(k)$ which is invoked every k turns). See Figure 9 for an illustration of the execution of such a machine.

The correctness of the algorithm can be established via `mWinFold_state`. In essence, it states that the `contentsff` and `contentsrr` together store the last k elements of the stream, and that the invariants on `aggsff` and `aggsrr` are maintained.

Theorem 17. Assume that elements of $\mathbb{V}$ can be stored in constant space and the lattice operations on $\mathbb{V}$ can be computed in constant time and space. Further, let $\varphi \in \Phi$ be a formula which only uses atomic functions which can be computed in constant time and space. Then, `toMonitor`(φ) is a Mealy machine whose state can be stored in $O(2^{|\varphi|})$ space and the transition (resp., output) functions `mNext` (resp., `mOut`) can be computed in amortized $O(|\varphi|)$ time per item.

Note: The exponential in the formula stems from the fact that the constants in the formula are encoded in binary. Note that this is unavoidable since computing the value of $\mathsf{P}_a p$ would require storing the last a values of p.

Proof (Theorem 17). This claim can be established via a straightforward induction on the structure of the formula φ. At each step in the induction, we need to show a constant space and amortized time overhead is created.

If φ is an atomic predicate, then computing φ can be done in constant time by assumption and it requires no additional state.

If $\varphi = \alpha \bullet \beta$ for $\bullet \in \{\wedge, \vee\}$, then we may assume by induction that `toMonitor`(α) (resp., `toMonitor`(β)) use $O(2^{|\alpha|})$ (resp., $O(2^{|\beta|})$) space and amortized $O(|\alpha|)$ (resp., $O(2^{|\beta|})$) time. The machine `toMonitor`$(\alpha * \beta)$ uses the states of both `toMonitor`(α) and `toMonitor`(β) and can be stored in $O(2^{|\alpha|} + 2^{|\beta|}) = O(2^{|\varphi|})$ space. By assumption, the additional time required to compute the lattice operation to combine the outputs of `toMonitor`(α) and `toMonitor`(β) is $O(1)$. So, this takes $O(|\alpha|) + O(|\beta|) + O(1) = O(|\varphi|)$ amortized time.

If $\varphi = X_{[0,\infty)}\alpha$ for $X \in \{\mathsf{P}, \mathsf{H}\}$ or $\alpha S_{[0,\infty)}\beta$, then the analysis is similar. In this case, the additional state we need to store is an element of $\mathbb{V}$, which we can

```
type ('v, 'a) formula =
| FAtomic of ('a -> 'v)
| FAnd of ('v, 'a) formula * ('v, 'a) formula
| FOr of ('v, 'a) formula * ('v, 'a) formula
| FSometime of int * int * ('v, 'a) formula
| FAlways of int * int * ('v, 'a) formula
| FSometimeUnbounded of int * ('v, 'a) formula
| FAlwaysUnbounded of int * ('v, 'a) formula
| FSince of int * int * ('v, 'a) formula * ('v, 'a) formula
| FSinceDual of int * int * ('v, 'a) formula * ('v, 'a) formula
| FSinceUnbounded of int * ('v, 'a) formula * ('v, 'a) formula
| FSinceDualUnbounded of int * ('v, 'a) formula * ('v, 'a) formula

type 'a lattice = { join : ('a -> 'a -> 'a); meet : ('a -> 'a -> 'a) }
type 'a boundedLattice = { bottom : 'a; top : 'a }

val toMonitor :
  'a1 lattice -> 'a1 boundedLattice
    -> ('a1, 'a2) formula -> ('a1, 'a2) monitor
```

Fig. 10. Extracted OCaml Code

store in $O(1)$ space. The additional time required is just a constant number of lattice operations, which can be done in $O(1)$ time. Thus, the inductive invariant is preserved in this case.

If $\varphi = X_a \alpha$ or $X_{[0,a]}\alpha$ for $X \in \{\mathsf{P}, \mathsf{H}\}$, then it is handled using a delay buffer or a sliding window aggregator as discussed. In both of these cases, a buffer of length $O(a)$ (i.e, $O(2^{|a|})$) is used. These queue mechanisms, as discussed above, are used in an "enqueue followed by dequeue" manner. The dequeue operations generally take $O(1)$ time but every a inputs involve reversal of the buffer which takes $O(a)$ time. This amounts to an amortized time of $O(1)$ per item.

4 Extraction and Experiments

We use Coq's extraction mechanism to produce OCaml code for our `toMonitor` function. This gives us a OCaml library the interface of which we show in Figure 10. The extracted `toMonitor` function can be instantiated with arbitrary bounded distributive lattices by specifying the operations $\sqcap$ and $\sqcup$ and the corresponding identities $\top$ and $\bot$.

For our experiments, we Following Example 5, we wish to emulate STL and use the lattice $(\mathbb{R} \cup \{\pm\infty\}, \max, \min)$. To do this, we model $\mathbb{R}$ with the concrete OCaml type `float`, which are 64-bit floating-point numbers. We also use $\mathbb{R}$ (modelled by `float`) for the set of data items $\mathbb{D}$. We compare the performance of our monitor with Reelay [59] (a C++ library) and the implementation for semiring-based monitoring algorithms in Rust from [46].

We have observed that the rate at which these tools process items roughly approaches a constant rate. Most notably, there are periodic spikes of latency that can be observed in our monitor, which correspond to the reversal of the lists in our queue based algorithms. A similar behavior is seen in the semiring-monitor, but this is harder to observe since the Rust implementation is very fast. We summarize performance using the amortized (i.e., average) time taken to process an item. To microbenchmark the building blocks of our algorithm, we consider formulas $X_{[0,n]}$, X_n, $X_{[n,2n]}$, $X_{[n,\infty)}$ where $X \in \{\mathsf{S}, \mathsf{P}\}$ and plot their performance with respect to n in Fig. 11. We notice that for Reelay, the performance depends on the type of input stream provided; so, we report our findings for stream whose elements are random, a stream whose elements form an increasing sequence and another which forms a decreasing sequence. In our tool and the semiring-monitor, the performance of the monitor seems to be roughly independent of the stream. Beyond certain values of the constants, some of the experiments with Reelay seemed to take prohibitively long time to process a given stream, preventing us from reliably measuring the performance at these values. Generally, we see that our tool has been performing better than Reelay but slower than the semiring-monitor, at least by an order of magnitude. We also see that the performance of our tool is roughly independent of the constants in the formula, as we expected from the analysis of our algorithms. The reported data is based on the mean of 6 trials of the experiments. The standard deviation is less than 15% of the mean in each case, and are indicated by whiskers.

A potential explanation for the comparative worse performance of Reelay is that Reelay stores data values in string-indexed maps. Interval Maps are also used in Reelay's implementation of operators such as $\mathsf{P}_{[\bullet,\bullet]}$. Since our tool does not use any map-like data structure, we do not incur these costs.

We have used the profiling tool Valgrind [57] to analyze the memory consumption of the monitors. In Fig. 11, we plot the peak memory usage of the monitors for the same formulas as before. For Reelay, we have reported the performance for three different traces. In the case of the semiring-monitor, the memory consumption can be explained near-perfectly with the help of the description of the algorithm (which is very similar to ours). This can be attributed to the fact that Rust programs have very minimal runtime overheads. The memory usage of Reelay is somewhat hard to understand, given that it is based on the Interval Maps data structures. Our tool is implemented in OCaml and its memory consumption is hard to interpret due to the garbage-collected nature of the language, however we do see a linear trend in the memory consumption with sufficiently high values for constants. The memory measurements for all three tools seemed to be deterministic, i.e, had the same value regardless of when it was executed.

In Figure 12, we use formulas inspired by the Timescales [58] benchmark to see how our tool performs when the constants in the formulae are scaled. The formulas used in the Timescales benchmark are in propositional MTL, so we define the propositions p, q, r and s as $x > 0.5$, $x > 0.0$, $x > 0.25$ and $x > 0.75$ respectively, where x is the value of the current sample in the trace.

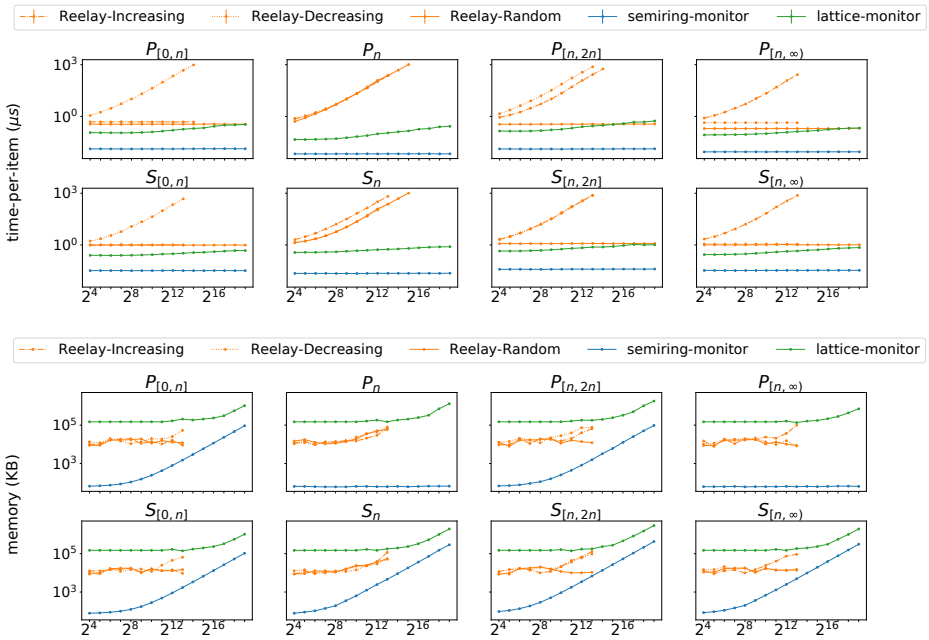

Fig. 11. Microbenchmarks: Formulas with large constants

For different values of n, the formulas F_0 through F_9 in Figure 12, in order, are: $\mathsf{H}(\mathsf{P}_{[0,n]}q \rightarrow (\neg p\ \mathsf{S}\ q))$, $\mathsf{H}(r \rightarrow \mathsf{P}_{[0,n]}(\neg p))$, $\mathsf{H}((r \wedge \neg q \wedge \mathsf{P}q) \rightarrow (\neg p\ \mathsf{S}_{[n,2n]}\ q))$, $\mathsf{H}(\mathsf{P}_{[0,n]}q \rightarrow (p\ \mathsf{S}\ q))$, $\mathsf{H}(r \rightarrow \mathsf{H}_{[0,k]}p)$, $\mathsf{H}((r \wedge \neg q \wedge \mathsf{P}q) \rightarrow (p\ \mathsf{S}_{[n,2n]}\ q))$, $\mathsf{HP}_{[0,n]}p$, $\mathsf{H}((r \wedge \neg q \wedge \mathsf{P}q) \rightarrow (\mathsf{P}_{[0,n]}(p \vee q)\ \mathsf{S}\ q))$, $\mathsf{H}((s \rightarrow \mathsf{P}_{n,2n}p) \wedge \neg(\neg s\ \mathsf{S}_{[n,\infty)}\ p))$, and $\mathsf{H}((r \wedge \neg q \wedge \mathsf{P}q) \rightarrow ((s \rightarrow \mathsf{P}_{[n,2n]}p) \wedge \neg(\neg s\ \mathsf{S}_{[n,\infty)}\ p)))$. Implications $\alpha \rightarrow \beta$ were encoded as $\neg\alpha \vee \beta$ and negations were encoded using their negation normal form. We have executed this experiment using traces with random values. The reported values are means of 10 trials, and we have used whiskers to denote the standard deviation.

All experiments were run on a computer with Intel Xeon CPUs 3.30GHz with 16 GB memory running Ubuntu 18.04.

5 Related Work

Fainekos and Pappas [31] introduce the notion of robustness for the interpretation of temporal formulas over discrete and continuous-time signals. In their setting, signals are represented as (time-dependent) functions that take value in a metric space. The distance function of the metric space is used to endow the signal space with a metric – the robustness is taken to be the largest extent to which a signal can be perturbed while still satisfying (or violating, depending on the case) the specification. In the same paper, an alternative quantitative semantics is proposed with an inductive definition that replaces disjunction with

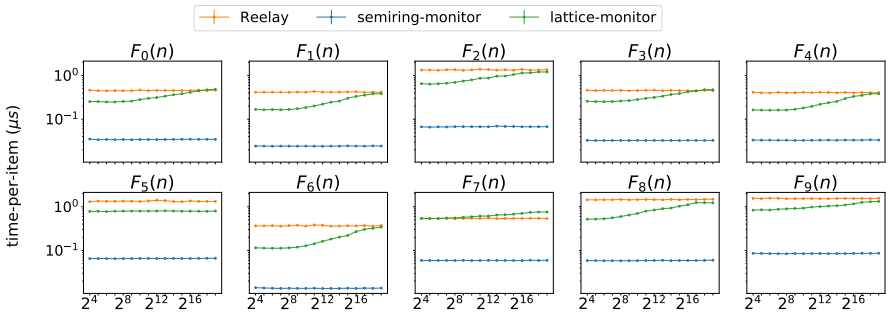

Fig. 12. Throughput for formulas from the Timescales benchmark

max and conjunction with min. This inductive semantics computes an under-approximation of the actual robustness value. This approach is extended by Donzé and Maler [27] to include temporal robustness. The inductive semantics of [31] can be computed efficiently, and forms the basis for the semantics we use.

The inductive semantics could be slightly generalized by interpreting conjunction (resp., disjunction) with multiplication (resp., addition) in some semiring. This idea subsumes the semantics of this paper since lattices are also semirings. In [46], this semantics is explored and an abstract version of the under-approximation guarantee from [31] is presented. With our approach, we would not be able to monitor formulas with this semantics since we make crucial use of the absorption laws in Lemma 14. In [46], a different approach for monitoring formulas with $\mathsf{S}_{[0,a]}$ is discussed that does not rely on this property. Our lattice-based semantics is considered in a dense-time setting in [47], along with a performance analysis of its Rust implementation.

The notion of distance between traces from [31] has been generalized in [37] to a more general algebraic setting using a semiring-based semantics. While both Mamouras et al. [46] and Jaksic et al. [37] consider truth domains that are semirings, the two works consider different semantics. Jaksic et al. suggest the use of symbolic weighted automata for the purpose of monitoring. With this approach, they are able to compute the precise robustness value for a property-signal pair.

The distance between two signals can be defined to be the maximum of the distance between the values that the signals take at corresponding points of time. However, other ways to define this distance have been considered. In [36], a quantitative semantics is developed via the notion of weighted edit distance. Averaging temporal operators are proposed in [4] with the goal of introducing an explicit mechanism for temporal robustness. The Skorokhod metric [23] has been suggested as a distance function between continuous signals. In [2], another metric is considered, which compares the value taken by the signal within a neighbourhood of the current time. Another interesting view of temporal logic is in [54], where temporal connectives are viewed as linear time-invariant filters.

Signal regular expressions (SREs) [60] are another formalism for describing patterns on signals. They are based on regular expressions, rather than LTL. SREs are a variant of the timed regular expressions (TREs) of [13], which are related to timed automata [5]. A robustness semantics for SRE has been proposed in [15] along with an algorithm for offline monitoring. In [14], STL is enriched by considering a more general (and quantitative) interpretation of the Until operator and adding specific aggregation operators.

In [43], a monitoring algorithm for STL is proposed and implemented in the AMT tool. A later version, AMT 2.0 [52] extends the capabilities of AMT to an extended version of STL along with TREs. In [26], an efficient algorithm for monitoring STL is discussed whose performance is linear in the length of the input trace. This is achieved by using Lemire's [41] sliding window algorithm for computing the maximum. This is implemented as a part the monitoring tool Breach [25]. A dynamic programming approach is used in [24] to design an online monitoring algorithm. Here, the availability of a predictor is assumed which predicts the future values, so that the future modalities may be interpreted. A different approach for online monitoring is taken in [28]: they consider robustness intervals, that is, the tightest interval which covers the robustness of all possible extensions of the available trace prefix. There are also monitoring formalisms that are essentially domain-specific languages for processing data streams, such as LOLA [22] and StreamQRE [48,8]. LOLA has recently been used as a basis for RtLOLA [33] in the StreamLAB framework [32], which adds support for sliding windows and variable-rate streams. A detailed survey on the many extensions to the syntax and semantics of STL along with their monitoring algorithms and applications is presented in [16].

In [20], a framework towards the formalization of runtime verification components are discussed. MonPoly [19] is a tool developed by Basin et al. aimed at monitoring a first order extension of temporal logic. In [55], the authors put forward Verimon, a simplified version of MonPoly which uses the proof assistant Isabelle/HOL to formally prove its correctness. They extend this line of work in Verimon+ [17] which verifies a more efficient version of the monitoring algorithms and uses a dynamic logic, which is an extension of the temporal logic with regular expression-like constructs. A verifying compiler for LOLA specifications to rust implementations has been developed in [35]. Their toolchain generates rust code from given LOLA specifications which are decorated with annotations that can be checked by the Viper [51] toolkit to verify functional correctness. On the one hand, a rust implementation would perform very well. On the other, the verification mechanism in this toolchain is based on SMT solvers while ours is based on the axioms of Coq's calculus, which is a much smaller system.

6 Conclusion

We have presented a formalization in the Coq proof assistant of a procedure for constructing online monitors for metric temporal logic with a quantitative semantics. We have extracted verified OCaml code from the Coq formalization.

Our experiments show that our formally verified online monitors perform well in comparison to Reelay [59], a state-of-the-art monitoring tool.

The construction of monitors that we presented can be extended and made more compositional by using classes of transducers that can support *dataflow combinators* [38] (serial, parallel and feedback composition), as seen in [49,45,50]. We leave an exploration of this direction as future work. It is also worth developing a more thorough benchmark suite to compare the presented monitoring framework against the tools Breach [25], S-TaLiRo [12], and StreamLAB [32]. We have extracted OCaml code from a Coq formalization, but a formally verified C implementation would be preferable from a performance standpoint. Another interesting direction is to increase the expressiveness of our specification formalism: one possible candidate is the extension to dynamic logic, as has been done in [17] in a qualitative setting.

Acknowledgments. This research was supported in part by US National Science Foundation award 2008096.

References

1. Abbas, H., Alur, R., Mamouras, K., Mangharam, R., Rodionova, A.: Real-time decision policies with predictable performance. Proceedings of the IEEE, Special Issue on Design Automation for Cyber-Physical Systems **106**(9), 1593–1615 (2018). https://doi.org/10.1109/JPROC.2018.2853608
2. Abbas, H., Mangharam, R.: Generalized robust MTL semantics for problems in cardiac electrophysiology. In: 2018 Annual American Control Conference (ACC). pp. 1592–1597. IEEE (2018). https://doi.org/10.23919/ACC.2018.8431460
3. Abbas, H., Rodionova, A., Mamouras, K., Bartocci, E., Smolka, S.A., Grosu, R.: Quantitative regular expressions for arrhythmia detection. IEEE/ACM Transactions on Computational Biology and Bioinformatics **16**(5), 1586–1597 (2019). https://doi.org/10.1109/TCBB.2018.2885274
4. Akazaki, T., Hasuo, I.: Time robustness in MTL and expressivity in hybrid system falsification. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 356–374. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_21
5. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**(2), 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8
6. Alur, R., Fisman, D., Mamouras, K., Raghothaman, M., Stanford, C.: Streamable regular transductions. Theoretical Computer Science **807**, 15–41 (2020). https://doi.org/10.1016/j.tcs.2019.11.018
7. Alur, R., Fisman, D., Raghothaman, M.: Regular programming for quantitative properties of data streams. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 15–40. Springer, Berlin, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49498-1_2
8. Alur, R., Mamouras, K.: An introduction to the StreamQRE language. Dependable Software Systems Engineering **50**, 1–24 (2017). https://doi.org/10.3233/978-1-61499-810-5-1

9. Alur, R., Mamouras, K., Stanford, C.: Automata-based stream processing. In: ICALP 2017. Leibniz International Proceedings in Informatics (LIPIcs), vol. 80, pp. 112:1–112:15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2017). https://doi.org/10.4230/LIPIcs.ICALP.2017.112
10. Alur, R., Mamouras, K., Stanford, C.: Modular quantitative monitoring. Proceedings of the ACM on Programming Languages **3**(POPL), 50:1–50:31 (2019). https://doi.org/10.1145/3290363
11. Alur, R., Mamouras, K., Ulus, D.: Derivatives of quantitative regular expressions. In: Aceto, L., Bacci, G., Bacci, G., Ingólfsdóttir, A., Legay, A., Mardare, R. (eds.) Models, Algorithms, Logics and Tools: Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday, LNCS, vol. 10460, pp. 75–95. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63121-9_4
12. Annapureddy, Y., Liu, C., Fainekos, G., Sankaranarayanan, S.: S-TaLiRo: A tool for temporal logic falsification for hybrid systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 254–257. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_21
13. Asarin, E., Caspi, P., Maler, O.: Timed regular expressions. Journal of the ACM **49**(2), 172–206 (2002). https://doi.org/10.1145/506147.506151
14. Bakhirkin, A., Basset, N.: Specification and efficient monitoring beyond STL. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11428, pp. 79–97. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17465-1_5
15. Bakhirkin, A., Ferrère, T., Maler, O., Ulus, D.: On the quantitative semantics of regular expressions over real-valued signals. In: Abate, A., Geeraerts, G. (eds.) FORMATS 2017. LNCS, vol. 10419, pp. 189–206. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65765-3_11
16. Bartocci, E., Deshmukh, J., Donzé, A., Fainekos, G., Maler, O., Ničković, D., Sankaranarayanan, S.: Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification, LNCS, vol. 10457, pp. 135–175. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_5
17. Basin, D., Dardinier, T., Heimes, L., Krstić, S., Raszyk, M., Schneider, J., Traytel, D.: A formally verified, optimized monitor for metric first-order dynamic logic. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS, vol. 12166, pp. 432–453. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51074-9_25
18. Basin, D., Klaedtke, F., Zalinescu, E.: Greedily computing associative aggregations on sliding windows. Information Processing Letters **115**(2), 186–192 (2015). https://doi.org/10.1016/j.ipl.2014.09.009
19. Basin, D., Klaedtke, F., Zalinescu, E.: The MonPoly monitoring tool. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 19–28. EasyChair (2017). https://doi.org/10.29007/89hs
20. Blech, J.O., Falcone, Y., Becker, K.: Towards certified runtime verification. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 494–509. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34281-3_34
21. Chlipala, A.: Certified programming with dependent types : a pragmatic introduction to the Coq proof assistant, chap. 5. The MIT Press, Cambridge, MA (2013). https://doi.org/10.5555/2584504
22. D'Angelo, B., Sankaranarayanan, S., Sanchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: Runtime monitoring of synchronous systems. In: 12th International Symposium on Temporal Representation and Reasoning (TIME 2005). pp. 166–174. IEEE (2005). https://doi.org/10.1109/TIME.2005.26

23. Deshmukh, J.V., Majumdar, R., Prabhu, V.S.: Quantifying conformance using the Skorokhod metric. Formal Methods in System Design **50**(2-3), 168–206 (2017). https://doi.org/10.1007/s10703-016-0261-8
24. Dokhanchi, A., Hoxha, B., Fainekos, G.: On-line monitoring for temporal logic robustness. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 231–246. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_19
25. Donzé, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 167–170. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_17
26. Donzé, A., Ferrère, T., Maler, O.: Efficient robust monitoring for STL. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 264–279. Springer, Heidelberg (2013)
27. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 92–106. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15297-9_9
28. Dreossi, T., Dang, T., Donzé, A., Kapinski, J., Jin, X., Deshmukh, J.V.: Efficient guiding strategies for testing of temporal properties of hybrid systems. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 127–142. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_10
29. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Van Campenhout, D.: Reasoning with temporal logic on truncated paths. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. pp. 27–39. Springer, Berlin, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_3
30. Fainekos, G., Hoxha, B., Sankaranarayanan, S.: Robustness of specifications and its applications to falsification, parameter mining, and runtime monitoring with s-taliro. In: Finkbeiner, B., Mariani, L. (eds.) Runtime Verification. pp. 27–47. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-32079-9_3
31. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. Theoretical Computer Science **410**(42), 4262–4291 (2009). https://doi.org/10.1016/j.tcs.2009.06.021
32. Faymonville, P., Finkbeiner, B., Schledjewski, M., Schwenger, M., Stenger, M., Tentrup, L., Torfah, H.: StreamLAB: Stream-based monitoring of cyber-physical systems. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 421–431. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_24
33. Faymonville, P., Finkbeiner, B., Schwenger, M., Torfah, H.: Real-time stream-based monitoring. CoRR **abs/1711.03829** (2017), `http://arxiv.org/abs/1711.03829`
34. Ferrère, T., Maler, O., Ničković, D., Pnueli, A.: From real-time logic to timed automata. Journal of the ACM **66**(3), 19:1–19:31 (2019). https://doi.org/10.1145/3286976
35. Finkbeiner, B., Oswald, S., Passing, N., Schwenger, M.: Verified rust monitors for lola specifications. In: Deshmukh, J., Ničković, D. (eds.) Runtime Verification. pp. 431–450. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-60508-7_24
36. Jakšić, S., Bartocci, E., Grosu, R., Nguyen, T., Ničković, D.: Quantitative monitoring of STL with edit distance. Formal Methods in System Design **53**(1), 83–112 (2018). https://doi.org/10.1007/s10703-018-0319-x

37. Jakšić, S., Bartocci, E., Grosu, R., Ničković, D.: An algebraic framework for runtime verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **37**(11), 2233–2243 (2018). https://doi.org/10.1109/TCAD.2018.2858460
38. Kahn, G.: The semantics of a simple language for parallel programming. Information Processing **74**, 471–475 (1974)
39. Kong, L., Mamouras, K.: StreamQL: A query language for processing streaming time series. Proceedings of the ACM on Programming Languages **4**(OOPSLA), 183:1–183:32 (2020). https://doi.org/10.1145/3428251
40. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Systems **2**(4), 255–299 (1990). https://doi.org/10.1007/BF01995674
41. Lemire, D.: Streaming maximum-minimum filter using no more than three comparisons per element. Nord. J. Comput. **13**(4), 328–339 (2006)
42. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) FTRTFT/FORMATS 2004. LNCS, vol. 3253, pp. 152–166. Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30206-3_12
43. Maler, O., Ničković, D.: Monitoring properties of analog and mixed-signal circuits. International Journal on Software Tools for Technology Transfer **15**(3), 247–268 (2013). https://doi.org/10.1007/s10009-012-0247-9
44. Maler, O., Nickovic, D., Pnueli, A.: Real time temporal logic: Past, present, future. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 2–16. Springer, Heidelberg (2005). https://doi.org/10.1007/11603009_2
45. Mamouras, K.: Semantic foundations for deterministic dataflow and stream processing. In: Müller, P. (ed.) ESOP 2020. LNCS, vol. 12075, pp. 394–427. Springer, Heidelberg (2020). https://doi.org/10.1007/978-3-030-44914-8_15
46. Mamouras, K., Chattopadhyay, A., Wang, Z.: Algebraic quantitative semantics for efficient online temporal monitoring. In: Groote, J.F., Larsen, K.G. (eds.) TACAS 2021. LNCS, vol. 12651, pp. 330–348. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_18
47. Mamouras, K., Chattopadhyay, A., Wang, Z.: A compositional framework for quantitative online monitoring over continuous-time signals. In: Feng, L., Fisman, D. (eds.) RV 2021. LNCS, vol. 12974, pp. 142–163. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88494-9_8
48. Mamouras, K., Raghothaman, M., Alur, R., Ives, Z.G., Khanna, S.: StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data. In: PLDI 2017. pp. 693–708. ACM (2017). https://doi.org/10.1145/3062341.3062369
49. Mamouras, K., Stanford, C., Alur, R., Ives, Z.G., Tannen, V.: Data-trace types for distributed stream processing systems. In: PLDI 2019. pp. 670–685. ACM, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314580
50. Mamouras, K., Wang, Z.: Online signal monitoring with bounded lag. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **39**(11), 3868–3880 (2020). https://doi.org/10.1109/TCAD.2020.3013053
51. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 41–62. Springer Berlin Heidelberg, Berlin, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
52. Ničković, D., Lebeltel, O., Maler, O., Ferrère, T., Ulus, D.: AMT 2.0: Qualitative and quantitative trace analysis with Extended Signal Temporal Logic. In:

Beyer, D., Huisman, M. (eds.) TACAS 2018. pp. 303–319. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_18
53. Pnueli, A., Zaks, A.: On the merits of temporal testers. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking: History, Achievements, Perspectives, LNCS, vol. 5000, pp. 172–195. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69850-0_11
54. Rodionova, A., Bartocci, E., Nickovic, D., Grosu, R.: Temporal logic as filtering. In: International Conference on Hybrid Systems: Computation and Control (HSCC 2016). pp. 11–20. ACM (2016). https://doi.org/10.1145/2883817.2883839
55. Schneider, J., Basin, D., Krstić, S., Traytel, D.: A formally verified monitor for metric first-order temporal logic. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 310–328. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32079-9_18
56. The Coq development team: The Coq proof assistant. `https://coq.inria.fr` (2021), [Online; accessed August 20, 2021]
57. The Valgrind Developers: Valgrind: An instrumentation framework for building dynamic analysis tools. `https://valgrind.org/` (2021), [Online; accessed August 20, 2021]
58. Ulus, D.: Timescales: A benchmark generator for MTL monitoring tools. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 402–412. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32079-9_25
59. Ulus, D.: The Reelay monitoring tool. `https://doganulus.github.io/reelay/` (2021), [Online; accessed August 20, 2021]
60. Ulus, D., Ferrère, T., Asarin, E., Maler, O.: Timed pattern matching. In: Legay, A., Bozga, M. (eds.) FORMATS 2014. LNCS, vol. 8711, pp. 222–236. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10512-3_16