# A Verified Online Monitor for Metric Temporal Logic with 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. The class of Moore 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

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.

The specification of temporal patterns is often driven by logical formalisms. Linear Temporal Logic (LTL) is one such widely utilized formalism which admits efficient algorithms. Since many applications in the domain of cyber-physical systems frequently deal with comparison between numerical signals, Signal Temporal Logic (STL), an extension of LTL with predicates allowing comparison with numerical values, is also popular. It is also common to constrain certain

temporal behaviors within specified time intervals, which is a capability referred to as a metric extension of temporal logic (MTL).

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 less than sufficiently informative: 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 apply some corrective actions if we could tell that our 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 [18].

The variant of metric temporal logic we consider in this paper is interpreted over a discrete temporal domain. We also consider 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 obligations has been considered, but these can be more expensive.

With Coq, an interactive theorem prover, 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.

As mentioned earlier, a strong motivation for using a quantitative semantics is to quantify how robustly a signal fits a given specification in view of potential perturbations. A very effective 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 [16]. 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 soon, does not hurt the performance of our algorithms.

In our formalization, we model online monitors as Moore machines. They are abstract machines whose state evolves as fragments of a trace are fed in.

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 Moore 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 Moore 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 Moore 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 Moore machine, (3) combinators which compute running aggregates on sliding windows, and (4) combinators which withhold the results of a machine until a given number of updates. We can see that most of these can be approached 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 [23]. 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.

*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 Moore machines, present a collection of Moore combinators, and discuss in detail their implementation. In Sect. 4, we discuss the extraction of executable OCaml code from the Coq scripts and we compare its performance against the Reelay tool [32]. 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, which will be the formalism that we consider here for specifying quantitative properties. We use bounded distributive lattices as the semantic domain for our logic. While this abstract setting is not usually where Metric Temporal Logic 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 $\top \in$ 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 $\infty$ 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*. A *trace* is a finite sequence over $\mathbb{D}$, and $\mathbb{D}^*$ is the set of all traces. We write $\varepsilon$ for the empty trace, and $|w|$ for the length of a trace $w \in \mathbb{D}^*$. For $i \in \mathbb{N}$ and $w \in D^*$ with $i \leq |w|$, we write $w[-i]$ to denote the prefix of $w$ obtained by removing the last $i$ elements of $w$. In particular, it holds that $w[-0] = w$ and $w[-i] = \varepsilon$ when $i = |w|$. We also fix a bounded distributive lattice $\mathbb{V}$, whose elements are *quantitative truth values* that represent degrees of truth or falsity. Our quantitative semantics will associate a truth value with each

formula-trace pair. The set $\Phi$ of temporal formulas that we consider are 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 $X \in \{\mathsf{P}, \mathsf{H}, \mathsf{S}\}$, 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 $\Phi$ over traces $\mathbb{D}^*$ using the *robustness* interpretation function $\rho : \Phi \times \mathbb{D}^* \to \mathbb{V}$, defined as follows:

$$\begin{aligned}
\rho(f, \varepsilon) &= \bot \\
\rho(f, w \cdot d) &= f(d), \text{ where } d \in \mathbb{D} \\
\rho(\varphi \vee \psi, w) &= \rho(\varphi, w) \sqcup \rho(\psi, w) \\
\rho(\varphi \wedge \psi, w) &= \rho(\varphi, w) \sqcap \rho(\psi, w) \\
\rho(\mathsf{P}_I \varphi, w) &= \bigsqcup_{\substack{i \in I \\ i < |w|}} \rho(\varphi, w[-i]) \\
\rho(\mathsf{H}_I \varphi, w) &= \bigsqcap_{\substack{i \in I \\ i < |w|}} \rho(\varphi, w[-i]) \\
\rho(\varphi \, \mathsf{S}_I \, \psi, w) &= \bigsqcup_{\substack{i \in I \\ i < |w|}} \Big( \rho(\psi, w[-i]) \sqcap \bigsqcap_{j < i} \rho(\varphi, w[-j]) \Big) \\
\rho(\varphi \, \overline{\mathsf{S}}_I \, \psi, w) &= \bigsqcap_{\substack{i \in I \\ i < |w|}} \Big( \rho(\psi, w[-i]) \sqcup \bigsqcup_{j < i} \rho(\varphi, w[-j]) \Big)
\end{aligned}$$

Notice that $\rho(\mathsf{P}_a \varphi, w) = \bot$ and $\rho(\mathsf{H}_a \varphi, w) = \top$ whenever $a \geq |w|$. Note that the temporal language that we consider does not include negation. However, this does not limit expressiveness as we discuss in the examples below.

**Example 6.** Extending Example 4, choose $\mathbb{D}$ to be $\mathbb{B}^k$ and 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 as equivalent 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 may 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 a NNF, again by 'pushing' negation inside while interchanging operators for their dual operators and replacing $r_i - c$ with $c - r_i$.

Since we are interpreting formulas over discrete traces, our logic is essentially equivalent to LTL with a "Previous" operator. In other words, temporal connectives (including $\mathsf{S}$ and $\overline{\mathsf{S}}$; see Lemma 13) with bounded intervals can be rewritten in terms of multiple compositions of the Previous operator instead.

## 3 The Monitoring Problem

Monitoring is analyzing a trace for specific 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 is fed in 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. We intend to discuss a mechanism for monitoring quantitative properties denoted by MTL formulas.

### 3.1 Moore Machines

We will use Moore machines, a class of sequence transducers, as a formal model of online monitoring algorithms.

**Definition 8.** Let $A$ and $B$ be sets. A *Moore 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, $\mathtt{mNext} : S \times A \to S$ is a transition function which transitions the state of the machine upon seeing an input from $A$, and $\mathtt{mOut} : S \to B$ associates an output with the current state. We write $\mathtt{Moore}(A, B)$ for the set of all Moore machines with inputs from $A$ and outputs from $B$.

While this is the standard definition of Moore Machines found in the literature, we use an equivalent, co-inductive definition in our formalization. In the co-inductive view, the states are not explicitly expressed, but described directly in terms of their extensional behavior.

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

The functions `mNext` and `mOut` denote the incremental update and the current output of the machine. More generally, the machine also associates with every trace a value, which can be simply obtained by feeding in the entire trace, element by element. In this sense, a machine denotes a quantitative property, formally captured in the definition of `gFinal` below.

**Definition 9.** Let $m \in \texttt{Moore}(A, B)$. Then, $\texttt{gNext}(m) : A^* \to \texttt{Moore}(A, B)$ is defined by $\texttt{gNext}(m, \varepsilon) = m$ and $\texttt{gNext}(m, w \cdot a) = \texttt{mNext}(\texttt{gNext}(m, w), a)$. $\texttt{gFinal}(m) : A^* \to B$ is defined as $\texttt{gFinal}(m, w) = \texttt{mOut}(\texttt{gNext}(m, w))$.

For a quantitative property of traces, i.e, a function $f : \mathbb{D}^* \to \mathbb{V}$, we wish to construct a Moore machine that computes $f$. We restrict our focus to quantitative properties which can be expressed by MTL formulas.

**Definition 10.** Let $\varphi \in \Phi$ and $m \in \texttt{Moore}(\mathbb{D}, \mathbb{V})$. We say that the Moore machine $m$ *implements a monitor for* $\varphi$ if $\texttt{gFinal}(m, w) = \rho(\varphi, w)$ for all $w \in \mathbb{D}^*$.

**Example 11.** Following Definition 8, consider the machine $m : \texttt{Moore}(\mathbb{V}, \mathbb{V})$ with states $\mathbb{V} \times \mathbb{V}$, initial state $(\bot, \bot)$, $\texttt{mOut}((u, v)) = u$ and $\texttt{mNext}((u, v), w) = (v, w)$. It holds that $\texttt{gFinal}(m, \varepsilon) = \texttt{gFinal}(m, v_1) = \bot$ and $\texttt{gFinal}(m, v_1 v_2) = v_1$, $\texttt{gFinal}(m, v_1 v_2 v_3) = v_2$, etc. The machine $m$ implements a monitor for the formula $\mathsf{P}_1(v \mapsto v)$ in the sense of Definition 10.

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

### 3.2 Moore 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.

Proceeding with the idea above, we identify the key constructs which are necessary in achieving the expressive power of MTL. We say that the formulas $\varphi$ and $\psi$ are *equivalent*, and we write $\varphi \equiv \psi$, if $\rho(\varphi, w) = \rho(\psi, w)$ for all $w \in \mathbb{D}^*$.

**Lemma 12.** The following identities hold:

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

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

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

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

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

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

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

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

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

**Lemma 13.** 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 with analogous arguments. Let $w \in \mathbb{D}^*$ be an arbitrary trace. We define $s_i = \rho(\varphi, w[-i])$ and $t_i = \rho(\psi, w[-i])$ for every $i \in \mathbb{N}$. Then, we have that

$$\rho(\varphi \,\mathsf{S}_{[0,a]}\, \psi, w) = \bigsqcup_{i \le K} \Big( t_i \sqcap \prod_{j<i} s_j \Big)$$

$$\rho(\varphi \,\mathsf{S}\, \psi, w) = \bigsqcup_{i \le |w|-1} \Big( t_i \sqcap \prod_{j<i} s_j \Big) = \rho(\varphi \,\mathsf{S}_{[0,a]}\, \psi, w) \sqcup \bigsqcup_{K < i \le |w|-1} \Big( t_i \sqcap \prod_{j<i} s_j \Big)$$

$$\rho(\mathsf{P}_{[0,a]}\psi, w) = \bigsqcup_{i \le K} t_i$$

where $K = \min(a, |w| - 1)$. We have to prove that $L = R \sqcap Q$, where $L = \rho(\varphi \,\mathsf{S}_{[0,a]}\, \psi, w)$, $R = \rho(\varphi \,\mathsf{S}\, \psi, w)$ and $Q = \rho(\mathsf{P}_{[0,a]}\psi, w)$. From $K \le |w| - 1$ we obtain that $L \sqsubseteq R$. It also holds that $L \sqsubseteq Q$ because $t_i \sqcap \prod_{j<i} s_j \sqsubseteq t_i$ for every $i \le K$. It follows that $L \sqsubseteq R \sqcap Q$. It remains to show that $R \sqcap Q \sqsubseteq L$. Since

$$\begin{aligned} R \sqcap Q &= \Big( L \sqcup \bigsqcup_{K < i \le |w|-1} \Big( t_i \sqcap \prod_{j<i} s_j \Big) \Big) \sqcap Q \\ &= (L \sqcap Q) \sqcup \bigsqcup_{K < i \le |w|-1} \Big( t_i \sqcap \prod_{j<i} s_j \sqcap Q \Big) \\ &= (L \sqcap Q) \sqcup \bigsqcup_{K < i \le |w|-1} \bigsqcup_{k \le K} \Big( t_i \sqcap t_k \sqcap \prod_{j<i} s_j \Big), \end{aligned}$$

it suffices to establish that $L \sqcap Q \sqsubseteq L$ (which is true) and $t_i \sqcap t_k \sqcap \prod_{j<i} s_j \sqsubseteq L$ for every $i$ and $k$ with $K < i \le |w| - 1$ and $k \le K$. Since $k < i$, we conclude that $t_i \sqcap t_k \sqcap \prod_{j<i} s_j \sqsubseteq t_k \sqcap \prod_{j<k} s_j \sqsubseteq L$. □

**Remark 14.** In the qualitative setting, the identities in Lemma 13 are intuitively clear, but they require a more careful argument in the quantitative setting. They have been used and proven in [15] 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 13, there is a subtlety in dealing with the terms of $\rho(\varphi \,\mathsf{S}\, \psi, w)$ with index $i = K + 1, \ldots, |w| - 1$.

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 $\mathsf{S}$ 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

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

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

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

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

$$\frac{m : \texttt{Moore}(\mathbb{D}, \mathbb{V}) \qquad k : \mathbb{N}}{\texttt{mSometimeWithin}\ k\ m : \texttt{Moore}(\mathbb{D}, \mathbb{V})} \qquad \frac{m : \texttt{Moore}(\mathbb{D}, \mathbb{V}) \qquad k : \mathbb{N}}{\texttt{mAlwaysWithin}\ k\ m : \texttt{Moore}(\mathbb{D}, \mathbb{V})}$$

**Fig. 1.** Summary of Moore Combinators

Boolean or temporal connective. The key observation is that this association between combinators on Moore machines and connectives *respect* the implementation relation (Definition 10) between machines and formulas. E.g., if $m$ is a monitor for $\varphi$, we expect `mSometimeWithin` $k\ m$ to be a monitor for $\mathsf{P}_{[0,k]}\varphi$.

We define the translation function $\texttt{toMonitor} : \Phi \to \texttt{Moore}(\mathbb{D}, \mathbb{V})$ as in Figure 2. We can think that $\texttt{toMonitor}(\varphi)$ is computed by recursively replacing the connectives in $\varphi$ with Moore combinators. The correctness of `toMonitor` is a consequence of the correctness of the combinators in the sense described above.

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. One example is the presence of dual connectives, such as $\vee$ and $\wedge$. This is why in many cases we focus on presenting these combinators in a slightly general way before instantiating them specifically to $\texttt{Moore}(\mathbb{D}, \mathbb{V})$. 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 `_final` which characterize the most recent output of the Moore machine at a fixed 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 Moore 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 $f : A \to B$ to $\texttt{Moore}(A, B)$, we define the `mLift` combinator, as in Figure 3. Given an $f : A \to B$ and a value $\texttt{init} : B$, it defines a Moore machine which computes $f$ on the latest input and initially emits `init`. We use the lemma `mLift_state` to describe the evolution of the machine when an arbitrary stream prefix is fed. Using this, we also prove `mLift_final`, which describes the final output of the machine after accepting

```
Fixpoint toMonitor {A : Type} (f : Formula) : Moore A Val :=
  match f with
  | FAtomic _ g => mAtomic g
  | FDelay _ n g => mDelay n (toMonitor g)
  | FDelayDual _ n g => mDelayDual n (toMonitor g)
  | FAnd _ g h => mAnd (toMonitor g) (toMonitor h)
  | FOr _ g h => mOr (toMonitor g) (toMonitor h)
  | FSometime _ g => mSometime (toMonitor g)
  | FAlways _ g => mAlways (toMonitor g)
  | FSince _  g h => mSince (toMonitor g) (toMonitor h)
  | FSinceDual _ g h => mSinceDual (toMonitor g) (toMonitor h)
  | FSometimeWithin _ hi g => mSometimeWithin hi (toMonitor g)
  | FAlwaysWithin _ hi g => mAlwaysWithin hi (toMonitor g)
  end.

Theorem toMonitor_correctness {A : Type} (f : Formula):
  implements (toMonitor f) f.
```

**Fig. 2.** The `toMonitor` function

an arbitrary stream prefix. We define `mAtomic` by instantiating the parameter `init` of `mLift` to $\bot$. 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 `gFinal`. 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 $\mathsf{P}_a$ and $\mathsf{H}_a$ as a mechanism that delays the output of a Moore Machine. For instance, the sequence $\langle \rho(\mathsf{P}_2\varphi, a_1a_2a_3), \rho(\mathsf{P}_2\varphi, a_1a_2a_3a_4)\rangle$ 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 `toMonitor`$(\varphi)$, enqueue the result and then return what we obtain by dequeuing. This works since the dequeued element was the result of `toMonitor`$(\varphi)$ $a$ turns ago. The queue needs to be initially filled with $a$ instances of $\bot$ (or $\top$ in the case of $\mathsf{H}_a$) 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

```
CoFixpoint mLift {A B: Type} (f : A -> B) (init : B) : Moore A B := {|
  mOut := init;
  mNext (a : A) := mLift f (f a) |}.
Lemma mLift_state {A B : Type}
  (xs : list A) (x : A) (f : A -> B) (init : B) :
    gNext (mLift f init) (xs ++ [x]) = mLift f (f x).
Lemma mLift_final {A B : Type}
 (xs : list A) (x : A) (f : A -> B) (init : B):
    gFinal (mLift f init) (xs ++ [x]) = f x.
Definition mAtomic {A : Type} (f : A -> Val) : Moore A Val :=
  mLift f bottom.
Lemma mAtomic_correctness {A : Type} (f : A -> Val):
  implements (mAtomic f) (FAtomic Val f).
```

**Fig. 3.** Establishing correctness of `mAtomic`.

work with. Functional lists are typically implemented via linked lists, which means that in order to access the $k$th 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 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 $\texttt{gCollect} : \texttt{Moore}(A, B) \times \mathbb{D}^* \to \mathbb{V}^*$ as

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

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

```
CoFixpoint mBinOp {A B C D: Type} (op : B -> C -> D)
    (m1: Moore A B) (m2 : Moore A C) : Moore A D :=
{|
  mOut := op (mOut m1) (mOut m2);
  mNext (a : A) := mBinOp op (mNext m1 a) (mNext m2 a)
|}.
Definition mAnd {A : Type} (m1 : Moore A Val)
  (m2 : Moore A Val) : Moore A Val := mBinOp meet m1 m2.
Definition mOr {A : Type} (m1 : Moore A Val)
  (m2 : Moore A Val) : Moore A Val := mBinOp join m1 m2.
Lemma mAnd_correctness {A : Type} (m1 m2 : Moore A Val) (f1 f2 : Formula):
  implements m1 f1 -> implements m2 f2
  -> implements (mAnd m1 m2) (FAnd Val f1 f2).
Lemma mOr_correctness {A : Type} (m1 m2 : Moore A Val) (f1 f2 : Formula):
  implements m1 f1 -> implements m2 f2
  -> implements (mOr m1 m2) (FOr Val f1 f2).
```

**Fig. 4.** The `mBinOp` combinator

```
Lemma delayWith_state {A B : Type} (init : B) (inf inb : list B)
      (m : Moore A B) (xs : list A) (x : A):
  forall initSeg, initSeg = [init] ++ inf ++ rev inb ->
  forall k, k = length initSeg ->
  forall str, str = initSeg ++ gCollect m xs ->
  forall lastSeg, lastSeg = lastn k str ->
    exists fr ba ii,
        [ii] ++ fr ++ rev ba = lastSeg /\
        k = length lastSeg /\
        gNext (delayWith init inf inb m) (xs ++ [x])
           = delayWith ii fr ba (gNext m (xs ++ [x])).
```

**Fig. 5.** Delay monitors.

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

Using the following identity, we may also view the computation of $\mathsf{S}$ 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.

```
CoFixpoint mFoldAux {A : Type} (m : Moore A B)
  (op : B -> B -> B) (st : B) : Moore A B :=
  {| mOut := st;
     mNext (a : A) := mFoldAux (mNext m a) (op st (mNextOut m a)) |}.
Definition mSometime {A : Type} (m : Moore A B) :=
  mFoldAux m meet bottom.
Definition mAlways {A : Type} (m : Moore A B) :=
  mFoldAux m join top.
```

**Fig. 6.** Temporal Folds

```
CoFixpoint mSinceAux {A : Type}
  (m1 m2 : Moore A Val) (pre : Val) : Moore A Val :=
  {| mOut := pre;
     mNext (a : A) :=
      mSinceAux (mNext m1 a) (mNext m2 a)
      (join (mNextOut m2 a) (meet (mNextOut m1 a) pre)) |}.
Definition mSince {A : Type} (m1 m2 : Moore A Val) :=
    mSinceAux m1 m2 (mOut m2).
```

**Fig. 7.** Monitoring Since

**Lemma 15.** 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 [13].

Using the equality of Lemma 15, `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 such a data structure can be found in [8].

Similar to the queues used in the delay monitors, `aggQueue` consists of two functional lists, the `rear` into which elements are inserted upon enqueing and the `front` out of which elements are evicted upon dequeing. The elements of `rear` and `front` are pairs: one of them is the enqueued element and the other represents a partial aggregate. For convenience, we denote by `contentsff`(resp., `contentsrr`) the enqueued elements currently in `front` (resp., `rear`) and by `aggsff`(resp., `aggsrr`) the partial aggregates currently in `front` (resp., `rear`). The aggregate values and the enqueued items are related in the following manner: (1) The $i$th-last element of `aggsff` is the aggregate of the last $i$ elements of

```
CoFixpoint mWinFoldAux {A : Type} (qq : aggQueue)
 (m : Moore A B) : Moore A B :=
  {| mOut := op (aggOut qq) (mOut m);
     mNext (a : A) :=
        mWinFoldAux (aggDQ (aggEnQ (mOut m) qq)) (mNext m a); |}.
Definition initAggQ (n : nat) : aggQueue :=
    {| front := repeat (unit, unit) (S n)
     ; rear := [] |}.
Definition mWinFold {A : Type} (m : Moore A B) (n : nat) : Moore A B :=
    mWinFoldAux (initAggQ n) m.
Lemma mWinFold_state {A : Type} (m : Moore A B)
 (n : nat) (xs : list A) (x : A) : exists qq,
  gNext (mWinFold m n) (xs ++ [x]) = mWinFoldAux qq (gNext m (xs ++ [x]))
  /\ contentsQ qq = lastn (S n) (repeat unit (S n) ++ gCollect m xs)
  /\ aggsffInv qq
  /\ aggsrrInv qq.
```

**Fig. 8.** Windowed Temporal Folds

`contentsff` (2) The $i$th-last element of `aggsrr` is the aggregate of the last $i$ elements of `contentsrr` taken in the reverse order. Given these invariants, it is easy to see that the aggregate of the entire queue can be computed as the aggregate of the heads of `aggsff` and `aggsrr`.

We maintain these invariants in the following way: Upon enqueue, we simply add the enqueued element to the head of `rear` along with the aggregate of the element with the head of `aggsff`. Performing a dequeue is easy when `front` is non-empty: we simply remove the element at its head. When `front` is empty, the contents of `contentsrr` are added to `front` while recalculating the aggregate, maintaining the invariant above.

Writing `op` for $\sqcup$ (resp., $\sqcap$) and `unit` for $\bot$ (resp., $\top$) in Figure 8, we define the combinator `mWinFold`. Given a constant $k$, `mWinFold` maintains an `aggQueue` initialized with $k$ instances of $\bot$ (or $\top$). When a new input is available, `mWinFold` enqueues the result of the corresponding submonitor into queue and dequeues the element which was enqueued $k$ turns ago. The output of `mWinFold` is simply set to be the aggregate of the elements in the queue. Using a similar argument as before, we can see that the invocations of `mNext`on `mWinFold` run in $O(1)$ amortized time. See Figure 9 for an illustration of the running of `mWinFold`.

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.

**Remark 16.** The space required by the described algorithm is constant in terms of the size of the input trace but exponential in the size of the constants that appear in the formula. This exponential is unavoidable since computing the value of $\mathsf{P}_a p$ would require storing the last $a$ values of $p$.

| `contentsff` | `aggsff` | `contentsrr` | `aggsrr` | `aggOut` |
|---|---|---|---|---|
| $\langle \bot, \bot, \bot \rangle$ | $\langle \bot, \bot, \bot \rangle$ | $\langle\rangle$ | $\langle\rangle$ | $\bot$ |
| $\langle \bot, \bot \rangle$ | $\langle \bot, \bot \rangle$ | $\langle a \rangle$ | $\langle a \rangle$ | $\bot \sqcup a$ |
| $\langle \bot \rangle$ | $\langle \bot \rangle$ | $\langle b, a \rangle$ | $\langle ab, a \rangle$ | $\bot \sqcup ab$ |
| $\langle\rangle$ | $\langle\rangle$ | $\langle c, b, a \rangle$ | $\langle abc, ab, a \rangle$ | $\bot \sqcup abc$ |
| $\langle b, c \rangle$ | $\langle bc, c \rangle$ | $\langle d \rangle$ | $\langle d \rangle$ | $bc \sqcup d$ |
| $\langle c \rangle$ | $\langle c \rangle$ | $\langle e, d \rangle$ | $\langle de, d \rangle$ | $c \sqcup de$ |

**Fig. 9.** A run of `mWinFold` while maintaining a queue of 3 elements. The elements $a, b, c, d, e$ are fed in, incrementally. The binary operation $\sqcup$ has been omitted in this figure except in a few places for the sake of brevity.

## 4 Extraction and Experiments

We use Coq's extraction mechanism to produce OCaml code for our `toMonitor` function. For this purpose, we instantiate the lattice $\mathbb{V}$ with the concrete OCaml type `float`.

We extract monitors for formulas involving atomic functions (involving projection and subtraction emulating STL, as explained in Example 5), Boolean operators and other temporal operators. As a measure of performance, we use throughput, which is the number of items that can be processed in a fixed duration. Since $\mathsf{P}_a$ and $\mathsf{P}_{[0,a]}$ are the main constructs used to express various other ones, we measure their performance for varying values of $a$ (see Figure 10). We also measure the throughput for monitors corresponding to similar formulas produced by Reelay [32].

We generate a trace consisting of three random floating point values in each trace item. For the purpose of our tool, we perform experiments on traces of length 20 million as well as 200 million. We observe that this difference on the length of the trace has no significant effect on the throughput. It appears that Reelay has a throughput which is slower by orders of magnitude. For this reason, we perform our experiments on Reelay on smaller traces - of 500 thousand items. The experiment corresponding to each formula was run 10 times and the reported value is their mean. The standard deviation of the results were less than 3% in all cases.

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 $\mathsf{P}_{[a,b]}$. Since our tool does not use any map-like data structure, we do not incur these costs.

In Figure 11, we use formulas similar to the ones used in the Timescales [31] benchmark. The formulas used in the Timescales benchmark are in propositional MTL, so we define the propositions $p$, $q$, $r$ and $s$ as $x > 0.5$, $y > 0.25$, $z > 0.3$ and $z > 0.6$ respectively, where $x$, $y$ and $z$ are projections of the current trace item. For convenience, also define $k$ and $\ell$ to be 1000 and 2000 respectively. The formulas F1 through F10 in Figure 11, in order, are: $\mathsf{H}(\mathsf{P}_{[0,k]}q \rightarrow (\neg p \mathsf{S} q))$, $\mathsf{H}(r \rightarrow$

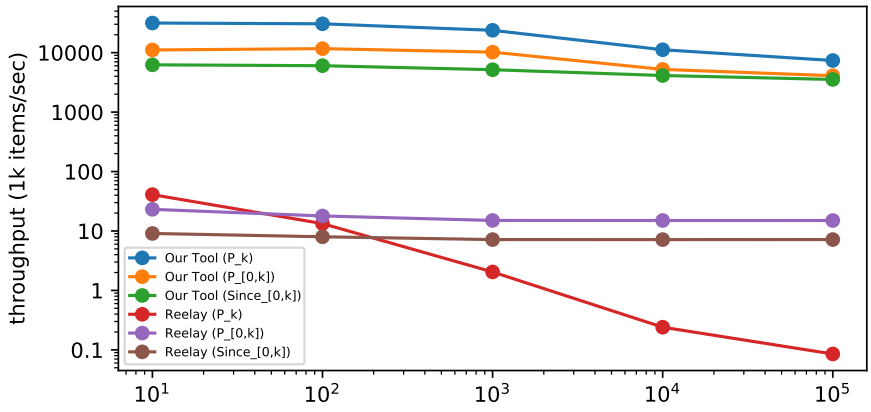


**Fig. 10.** Throughput for formulas with large constants

$\mathsf{P}_{[0,k]}(\neg p)$, $\mathsf{H}((r \wedge \neg q \wedge \mathsf{P} q) \rightarrow (\neg p\, \mathsf{S}_{[k,\ell]}\, q))$, $\mathsf{H}(\mathsf{P}_{[0,k]} q \rightarrow (p\, \mathsf{S}\, q))$, $\mathsf{H}(r \rightarrow \mathsf{H}_{[0,k]} p)$, $\mathsf{H}((r \wedge \neg q \wedge \mathsf{P} q) \rightarrow (p \mathsf{S}_{[k,\ell]}\, q))$, $\mathsf{H}\mathsf{P}_{[0,k]} p$, $\mathsf{H}((r \wedge \neg q \wedge \mathsf{P} q) \rightarrow (\mathsf{P}_{[0,k]}(p \vee q) \mathsf{S} q))$, $\mathsf{H}((s \rightarrow \mathsf{P}_{k,\ell} p) \wedge \neg(\neg s\, \mathsf{S}_{[k,\infty)}\, p))$, and $\mathsf{H}((r \wedge \neg q \wedge \mathsf{P} q) \rightarrow ((s \rightarrow \mathsf{P}_{[k,\ell]} p) \wedge \neg(\neg s\, \mathsf{S}_{[k,\infty)}\, p)))$. Implications $\alpha \rightarrow \beta$ were encoded as $\neg\alpha \vee \beta$ and negations were encoded using their negation normal form.

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 [18] introduce the notion of robustness for the interpretation of temporal formulas over discrete and continuous time signals. In their setting, signals are functions from a temporal domain to a metric space and the distance function of the metric space is used to endow the space of signals with a metric. The robustness is essentially the largest amount by which a signal can be perturbed while still satisfying the specification. In the same paper, an alternate quantitative semantics is proposed which is defined in an inductive fashion by replacing disjunction with max and conjunction with min. This inductive semantics can be used to under-approximate the robustness value. The framework used in our paper essentially is this under-approximating semantics. This approach is extended by Donzé and Maler [16] to include temporal robustness.

In [21], the authors describe a general algebraic framework for defining robustness based on the monoidal structure of traces using the semiring structure on the semantic domain. They 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 construction of a weighted automaton from a temporal formula incurs a doubly exponential blowup, if one

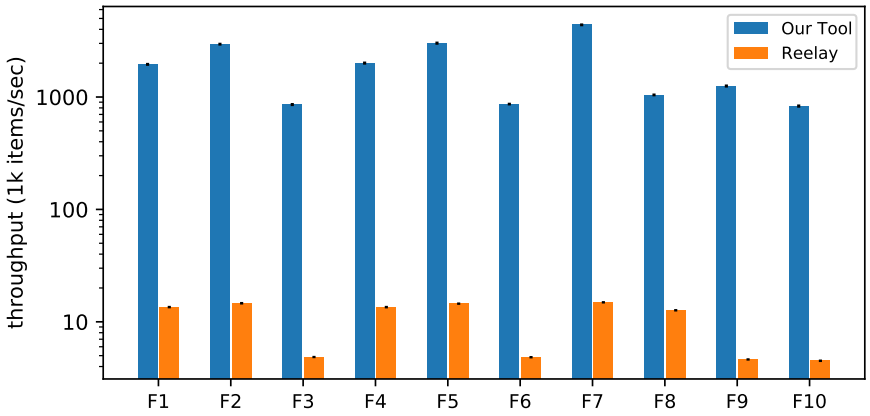


**Fig. 11.** Throughput for formulas from the Timescales benchmark

assumes a succinct binary representation of the constants appearing in an MTL formula. We consider here a class of lattices, which are also semirings. With our approach, however, we do not calculate the precise robustness value, but an under-approximation in the sense discussed in the previous paragraph. One may also consider a semantics in which disjunction is replaced with $+$ and conjunction with $\times$ from the semiring. 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 13. The most interesting semiring which does not form a lattice which might be relevant in the monitoring of cyber-physical systems is $(\mathbb{R}, \max, +)$.

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 [20], a quantitative semantics is developed via the notion of weighted edit distance. Averaging temporal operators are proposed in [2] with the goal of introducing an explicit mechanism for temporal robustness. The Skorokhod metric [12] has been suggested as a distance function between continuous signals. In [1], 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 [29], where temporal connectives are viewed as linear time-invariant filters.

Signal Regular Expressions (SREs) [33] are another formalism for describing patterns on signals. They are based on regular expressions, rather than LTL. A robustness semantics for SRE has been proposed in [5] along with an algorithm for offline monitoring. In [4], STL is enriched by considering a more general (and quantitative) interpretation of the Until operator and adding specific aggregation operators. They also give a semantics of their formalism using dual numbers, which are the real numbers with an adjoined element $\epsilon$ satisfying $\epsilon^2 = 0$.

In [24], a monitoring algorithm for STL is proposed and implemented in the AMT tool. A later version, AMT 2.0 [28] extends the capabilities of AMT to an extended version of STL along with Timed Regular Expressions. In [15], 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 [23] sliding window algorithm for computing the maximum. This is implemented as a part the monitoring tool Breach [14]. A dynamic programming approach is used in [13] 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 towards online monitoring is taken in [17]: 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 [11] and StreamQRE [26]. LOLA has recently been used as a basis for RtLOLA in the StreamLAB framework [19], 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 [6].

In [10], a framework towards the formalization of runtime verification components are discussed. MonPoly [9] is a tool developed by Basin et al. aimed at monitoring a first order extension of temporal logic. In [30], 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+ [7] 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.

## 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 [32], 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* [22] (serial, parallel and feedback composition), as seen in [25, 27]. 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 [14], S-TaLiRo [3], and StreamLAB [19]. 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 [7] in a qualitative setting.

## References


1. Abbas, H., Mangharam, R.: Generalized robust MTL semantics for problems in cardiac electrophysiology. In: ACC 2018. pp. 1592–1597. IEEE (2018)
2. 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)
3. 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)
4. 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)
5. 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)
6. 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)
7. 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)
8. Basin, D., Klaedtke, F., Zalinescu, E.: Greedily computing associative aggregations on sliding windows. Information Processing Letters **115**(2), 186–192 (2015)
9. 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)
10. 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)
11. 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: TIME 2005. pp. 166–174. IEEE (2005)
12. 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)
13. 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)
14. 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)
15. 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)
16. 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)

17. 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)
18. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. Theoretical Computer Science **410**(42), 4262–4291 (2009)
19. 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)
20. 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)
21. 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)
22. Kahn, G.: The semantics of a simple language for parallel programming. Information Processing **74**, 471–475 (1974)
23. Lemire, D.: Streaming maximum-minimum filter using no more than three comparisons per element. Nord. J. Comput. **13**(4), 328–339 (2006)
24. 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)
25. 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)
26. 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)
27. Mamouras, K., Wang, Z.: Online signal monitoring with bounded lag (2020), accepted for publication in the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, ESWEEK-TCAD special issue (EMSOFT 2020)
28. 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)
29. 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)
30. 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)
31. 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)
32. Ulus, D.: The Reelay monitoring tool. https://doganulus.github.io/reelay/ (2020), [Online; accessed August 20, 2020]
33. 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)