Striver: Stream Runtime Verification for Real-Time Event-Streams *

Felipe Gorostiaga and César Sánchez

IMDEA Software Institute, Spain
{felipe.gorostiaga,cesar.sanchez}@imdea.org

Abstract. We study the problem of monitoring rich properties of real-time event streams, and propose a solution based on Stream Runtime Verification (SRV), where observations are described as output streams of data computed from input streams of data. SRV allows a clean separation between the temporal dependencies among incoming events, and the concrete operations that are performed during the monitoring.
SRV specification languages typically assume that all streams share a global synchronous clock and input events arrive in a synchronous manner. In this paper we generalize the time assumption to cover real-time event streams, but keep the essential explicit time dependencies present in synchronous SRV languages. We introduce Striver, which shares with SRV the simplicity, and the separation between the timing reasoning and the data domain. Striver is a general language that allows to express other real-time monitoring languages. We show in this paper translations from other formalisms for (piece-wise constant) signals and timed event streams. Finally, we report an empirical evaluation of an implementation of Striver.

This is a post-peer-review, pre-copyedit version of an article published at LNCS vol 11237, Springer (2018).The final authenticated version is available online at: https://doi.org/10.1007/978-3-030-03769-7_16.

1 Introduction

Runtime verification (RV) is a lightweight formal method that studies the problem of whether a single trace from the system under analysis satisfies a formal specification. From the point of view of coverage, static verification must consider all possible executions of the system while RV only considers the traces observed. In this manner, RV sacrifices completeness but offers a readily applicable formal method that can be combined with testing or debugging. See [16,20] for surveys on RV, and the recent book [2]. Early specification languages proposed in RV

* This research has been partially supported by: the EU H2020 project Elastest (num. 731535), by the Spanish MINECO Project "RISCO (TIN2015-71819-P)" and by the EU ICT COST Action IC1402 ARVI (*Runtime Verification beyond Monitoring*).

were based on temporal logics [17,12,7], regular expressions [25], timed regular expressions [3], rules [5], or rewriting [24].

Stream runtime verification (SRV), pioneered by Lola [11] defines monitors by declaring the dependencies between output streams (results) and input streams (observations). The main idea of SRV is that the same sequence of operations performed during the monitoring of a temporal logic formula can be followed to compute statistics of the input trace, if the data type and the operations are changed. The generalization of the outcome of the monitoring process to richer verdict values brings runtime verification closer to monitoring and data stream-processing. See [22,15,8] for further works on SRV. Temporal testers [23] were later proposed as a monitoring technique for LTL based on Boolean streams. SRV was initially conceived for monitoring synchronous systems, where all observations proceed in cycles. In this paper we present a specification formalism for timed asynchronous observations, where streams are sequences of timed events, not necessarily happening at the same time in all input streams, but where all time-stamps are totally ordered according to a global clock (following the timed asynchronous model of distributed systems [10]). The formalism that we propose in this paper targets the outline, non-intrusive monitoring (see [18] for definitions), where the model of time is that of timed asynchronous distributed systems. Our target application is the monitoring and testing of cloud systems and multi-core hardware monitoring, where this assumption is reasonable.

Related work The work [19] presents an asynchronous evaluation engine for a simple event stream language for timed events, based on a collection of language constructs that compute aggregations. This language does not allow explicit time references and offsets. Moreover, recursion is not permitted and all recursive computations are encapsulated implicitly in the language constructs. A successor work of [19] is TeSSLa [9] which allows recursion and offers a smaller collection of language constructs. Still, TeSSLa precludes explicit offset dependencies, and the target application domain is hardware based monitoring. We sketched that Striver subsumes TeSSLa. Another similar work is RTLola [14], which also aims to extend SRV from the synchronous domain to timed streams. However, in RTLola defined streams are computed at predefined periodic instants of time, collecting aggregations between these predefined instants using language constructs. In this manner, the output streams in RTLola are *isochronous*[1], while in Striver defined streams are computed at the specific real-time instants where they are required, resulting in a completely *asynchronous* SRV system (in the sense that streams can tick at arbitrary time points). Striver can be used as a low level language to compile TeSSLa, RTLola and similar specifications.

The rest of the paper is organized as follows. Section 2 describes the Striver specification language. Section 3 presents a trace-lenght independent online algorithm. Section 4 shows some extensions of Striver. Section 5 reports on an empirical evaluation and Section 6 concludes the paper.

[1] We borrow this term from telecomunications and signal processing where an isochronous signal is one in which events happen at regular intervals.

2 The Striver Specification Language

In this section we introduce Striver, a specification language that allows defining efficiently monitorable specifications [11], those for which all streams can be resolved immediately. We show in Section 3 an online monitoring algorithm and prove that this algorithm is also trace length independent.

2.1 Preliminaries

The main idea behind SRV is to separate two concerns: the temporal dependencies and the data manipulated, for which we use data domains.

Data Domains. We use many-sorted first order logic to describe data domains. A simple theory, *Booleans*, has only one sort[2], *Bool*, two constants `true` and `false`, binary functions $\wedge$ and $\vee$, unary function $\neg$, etc. A more sophisticated signature is *Naturals* that consists of two sorts (*Nat* and *Bool*), with constant symbols `0`, `1`, `2`... of sort *Nat*, binary symbols $+$, $*$, etc (of sort $\mathit{Nat} \times \mathit{Nat} \to \mathit{Nat}$) as well as predicates $<$, $\leq$, etc of sort $\mathit{Nat} \times \mathit{Nat} \to \mathit{Bool}$, with their usual interpretation. All theories have equality and are typically (e.g. *Naturals*, *Booleans*, *Queues*, *Stacks*, etc) equipped with a ternary symbol $\texttt{if} \cdot \texttt{then} \cdot \texttt{else}\cdot$. In the case of *Naturals*, the $\texttt{if} \cdot \texttt{then} \cdot \texttt{else}\cdot$ symbol has sort $\mathit{Bool} \times \mathit{Nat} \times \mathit{Nat} \to \mathit{Nat}$.

Our theories are interpreted, so each sort S is associated with a domain D_S (a concrete set of values), and each function symbol `f` is interpreted as a total computable function f, with the given arity and that produces values of the domain of the result given elements of the arguments' domains. For simplicity, we omit the sort S from D_S.

We will use *stream variables* with an associated sort, but from the point of view of the theories, these stream variables are atoms. As usual, given a set of sorted atoms A and a theory, the set of terms is the smallest set containing A and closed under the use of function symbols in the theory as a constructors (respecting sorts).

We consider a special *time* domain $\mathbb{T}$, whose interpretation is a (possibly infinite, possibly dense) set with a total order and a minimal element 0, and a binary addition symbol $+$. Examples of time domains are $\mathbb{R}_0^+$, $\mathbb{Q}_0^+$ and $\mathbb{N}_0$ with their usual order. Given $t_a, t_b \in \mathbb{T}$ we use $[t_a, t_b] = \{t \in \mathbb{T} \mid t_a \leq t \leq t_b\}$, and also (t_a, t_b), $[t_a, t_b)$ and $(t_a, t_b]$ with the usual meaning. We say that a set of time points $S \subseteq \mathbb{T}$ does not contain bounded infinite subsets, whenever for every $t_a, t_b \in \mathbb{T}$, the set $S \cap [t_a, t_b]$ is finite, in which case we say that S is a non-Zeno set.

We extend every domain D into $D^\perp$ that includes two special fresh symbols $\perp^D_{\text{notick}}$ and $\perp^D_{\text{outside}}$. These new symbols allow capturing when a stream does not generate an event, and when the time offset falls off the beginning and the end of the trace.

[2] We use sort and type interchangeably in the rest of the paper.

Streams. Monitors observe sequences of events as inputs, where each event is time-stamped and contains a data value from its domain.

Definition 1 (Event stream). *An event stream of sort D is a partial function $\eta : \mathbb{T} \rightharpoonup D$ such that $dom(\eta)$ does not contain bounded infinite subsets, where $dom(\eta)$ is the subset of $\mathbb{T}$ where η is defined.*

The set $dom(\eta)$ is called the set of *event points* of η. An event stream η can be naturally represented as a *timed word*: $s_\eta = (t_0, \eta(t_0))(t_1, \eta(t_1)) \cdots \in (dom(\eta) \times D)^*$, or as an ω-timed word $s_\eta = (t_0, \eta(t_0))(t_1, \eta(t_1)) \cdots \in (dom(\eta) \times D)^\omega$ for infinite streams, such that:
(1) s_η is ordered by time ($t_i < t_{i+1}$); and
(2) for every $t_a, t_b \in \mathbb{T}$ the set $\{(t, d) \in s_\eta \mid t \in [t_a, t_b]\}$ is finite.
The set of all event streams over D is denoted by $\mathcal{E}_D$.

We introduce some notation for event streams. The functions $prev_<$ and $prev_\leq$ with type $\mathcal{E}_D \times \mathbb{T} \to \mathbb{T}^\perp$ are defined as follows. Note that the functions can return a value in $\mathbb{T}^\perp$ because *sup* can return $\perp^{\mathbb{T}}_{\text{outside}}$ when the stream has no event in the interval provided.

$$prev_<(\sigma, t) \stackrel{\text{def}}{=} sup(dom(\sigma) \cap [0, t)) \qquad prev_\leq(\sigma, t) \stackrel{\text{def}}{=} sup(dom(\sigma) \cap [0, t])$$

$$sup(S) \stackrel{\text{def}}{=} \begin{cases} max(S) & \text{if } S \neq \emptyset \\ \perp^{\mathbb{T}}_{\text{outside}} & \text{otherwise} \end{cases}$$

Essentially, given a stream σ and a time instant $t \in \mathbb{T}$, the expression $prev_<(\sigma, t)$ provides the nearest time instant in the past of t at which σ is defined. Similarly, $prev_\leq(\sigma, t)$ returns t if $t \in dom(\sigma)$, otherwise it behaves as $prev_<$.

Synchronous SRV In synchronous SRV, specifications are given by associating every output stream variable y with a defining equation that, once the input streams are known, associates y to an output stream. For example:

```
define bool always_p := p /\ always_p[-1,true]
define int   count_p := (count_p[-1,0]) + if p then 1 else 0
```

defines two output streams: `always_p`, which calculates whether Boolean input stream p was true at every point in the past (that is, $\boxminus p$) and `count_p`, which counts the number of times p was true in the past. Offset expressions like `count_p[-1,0]` allow referring to streams in a different position (in this case in the previous position) with a default value when there is no previous position (the beginning of the trace). In this paper we introduce a similar formalism for timed event streams. Our goal is to provide a simple language with few constructs including explicit references to the previous position at which some stream contains an event, contrary to other stream languages like TeSSLa [9] and RTLola [14] which preclude to reason about real-time instants. We say that Striver is an *explicit time* SRV formalism.

2.2 Syntax of Striver

A Striver specification describes the relation between input event-streams and output event-streams, where an input stream is a sequence of observations from the system under analysis.

> The key idea in Striver is to associate each defined stream variable with:
> – a *ticking expression* that captures when the stream may contain an event;
> – a *value expression* that defines the value contained in the event.

Note that in synchronous SRV, only a value expression is necessary because every stream has a value at every clock tick.

Formally, a Striver specification $\varphi : \langle I, O, V, T \rangle$ consists of input stream variables $I = \{x_1, \ldots, x_n\}$, output stream variables $O = \{y_1, \ldots, y_m\}$, a collection of ticking expressions $T = \{T_1, \ldots, T_m\}$ and a collection of value expressions $V = \{V_1, \ldots, V_m\}$. For output variable y, T_y captures when stream y ticks and V_y what the value is when y ticks. All input and output streams are associated with a sort. It is sometimes convenient to partition output streams into proper outputs and intermediate streams, that are introduced only to simplify specifications.

In practice, it is very useful that T_y defines an over-approximation of the set of instants at which y ticks, and then allow the value expression to evaluate to $\bot^D_{\text{notick}}$. The stream associated with y does not contain an event at t if V_y evaluates to $\bot^D_{\text{notick}}$ at t, even if t is in T_y. For example, if one wishes y to filter out events from a given stream x it is simple to define in T_y that y ticks whenever x does, and delegate to V_y to decide whether an event is relevant of should be filtered out.

Expressions. We fix a set of stream variables $Z = I \cup O$. Apart from ticking expressions and value expressions, offset expressions (used inside value expressions) allow defining temporal dependencies between ticking instants.

– *Ticking Expressions*:

$$\alpha := \{c\} \mid v.\texttt{ticks} \mid \alpha \,\mathsf{U}\, \alpha \mid \texttt{delay}\ w$$

where $c \in \mathbb{T}$ is a time constant, v is an arbitrary stream variable, and w is a stream variable of type $\mathbb{T}_\epsilon$, and U is used for the union of sets of ticks. The type $\mathbb{T}_\epsilon$ is defined as $\mathbb{T}_\epsilon = \{t \mid t \geq \epsilon\}$ for a given $\epsilon > 0$. This restriction on the argument of `delay` guarantees that the ticking instants are non-zeno if all their inputs are non-zeno (see Section 3).

– *Offset Expressions*, which allow fetching previous events from streams:

$$\tau_x ::= x \mathbin{<\!\!\sim} \tau' \mid x \ll \tau' \qquad \tau' ::= \texttt{t} \mid \tau_z \text{ for } z \in Z$$

Offset expressions have sort $\mathbb{T}$. Here, `t` represents the current value of the clock. The intended meaning of $x \ll \tau'$ is to refer to the previous instant

strictly in the past of τ' where x ticks (or $\bot^D_{\text{outside}}$ if there is not such an instant). The expression x `<~` τ' also considers the present as a candidate.

– *Value Expressions*, which give the value of a defined stream at a given ticking point candidate:

$$E ::= d \mid x(\tau_x) \mid \mathtt{f}(E_1, \ldots, E_k) \mid \mathtt{t} \mid \tau_x \mid \mathtt{outside}_D \mid \mathtt{notick}_D$$

where d is a constant of type D, $x \in Z$ is a stream variable of type D and `f` is a function symbol of return type D. Note that in $x(\tau_x)$ the value of stream x is fetched at an offset expression indexed by x, which captures the ticking points of x and guarantees the existence of an event. Expressions `t` and τ_x build expressions of sort $\mathbb{T}$. The two additional constants $\mathtt{outside}_D$ and $\mathtt{notick}_D$ allow to reason about accessing the end of the streams, or not generating an event at ticking candidate instant.

We also use the following syntactic sugar:

$$x(\mathtt{\tilde{}}e) \stackrel{\text{def}}{=} x(x\mathtt{<\tilde{}}\ e) \qquad x(\mathtt{\tilde{}}e, d) \stackrel{\text{def}}{=} \mathtt{if}\ (x\ \mathtt{<\tilde{}}\ e)\mathtt{==outside\ then}\ d\ \mathtt{else}\ x(\mathtt{\tilde{}}e)$$
$$x(\mathtt{<}e) \stackrel{\text{def}}{=} x(x\ \mathtt{<<}\ e) \qquad x(\mathtt{<}e, d) \stackrel{\text{def}}{=} \mathtt{if}\ (x\ \mathtt{<<}\ e)\mathtt{==outside\ then}\ d\ \mathtt{else}\ x(\mathtt{<}e)$$

Essentially, x`(~t)` provides the value of x at the previous ticking instant of x (including the present) and x`(<t)` is similar but not including the present. Also, x`(<t,`d`)` is the analogous to $x[-1, d]$ in synchronous SRV allowing to fetch the value in the previous event in stream x, or d if there is not such previous event.

Example 1. Consider two input event streams: `sale` that represents sales of a certain product, and `arrival` which represents the arrivals to the store:

```
input int sale, int arrival
ticks stock := sale.ticks U arrival.ticks
define int stock := stock(<t,0) +
    (if isticking(arrival) then arrival(~t) else 0) -
    (if isticking(sale   ) then sale(~t   ) else 0)
```

where `isticking(sale)` is defined as `(sale<~t)==t`. Note that `stock` is defined to tick when either `sale` or `arrival` (or both) tick. □

Example 2. To illustrate the use of `delay` consider the following specification:

```
ticks clock       := {0} U delay clock
define Time_eps clock := 1sec
```

The stream `clock` emits an event every second since time 0. □

2.3 Semantics

As common in SRV, the semantics is defined denotationally first. This semantics establishes whether a given input and a given output satisfy the specification, which is defined in terms of *valuations*. Given a set of variables Z, a valuation

σ is a map that associates every x in Z of sort D with an event stream from $\mathcal{E}_D$. Given a valuation σ we define the result of evaluating an expression for σ. We define three evaluation maps $[\![.]\!]_\sigma$, $[\![.]\!]_\sigma$, $[\![.]\!]_\sigma$ depending on the type of the expression[3]:

– *Ticking Expressions.* The semantic map $[\![.]\!]_\sigma$ assigns a set of time instants to each ticking expression as follows:

$$\begin{aligned}
[\![\{c\}]\!]_\sigma &\stackrel{\text{def}}{=} \{c\} \\
[\![v.\texttt{ticks}]\!]_\sigma &\stackrel{\text{def}}{=} dom(\sigma_v) \\
[\![a \,\mathsf{U}\, b]\!]_\sigma &\stackrel{\text{def}}{=} [\![a]\!]_\sigma \cup [\![b]\!]_\sigma \\
[\![\texttt{delay}(w)]\!]_\sigma &\stackrel{\text{def}}{=} \{t' \mid \text{ there is a } t \in dom(\sigma_w) \text{ such that } t + \sigma_w(t) = t' \\
&\qquad \text{and } dom(\sigma_w) \cap (t, t') = \emptyset \,\}
\end{aligned}$$

– *Offset Expressions.* For offset expressions $[\![.]\!]_\sigma$ provides, given a time instant t, another time instant:

$$\begin{aligned}
[\![\texttt{t}]\!]_\sigma(t) &\stackrel{\text{def}}{=} t \\
[\![x \texttt{ << } e]\!]_\sigma(t) &\stackrel{\text{def}}{=} \begin{cases} \bot^{\mathbb{T}}_{\text{outside}} & \text{if } [\![e]\!]_\sigma(t) = \bot^{\mathbb{T}}_{\text{outside}} \\ prev_<(\sigma_x, [\![e]\!]_\sigma(t)) & \text{otherwise} \end{cases} \\
[\![x \texttt{ <~ } e]\!]_\sigma(t) &\stackrel{\text{def}}{=} \begin{cases} \bot^{\mathbb{T}}_{\text{outside}} & \text{if } [\![e]\!]_\sigma(t) = \bot^{\mathbb{T}}_{\text{outside}} \\ prev_\leq(\sigma_x, [\![e]\!]_\sigma(t)) & \text{otherwise} \end{cases}
\end{aligned}$$

– *Value Expressions.* Finally, value expressions are evaluated into event streams of the appropriate type. For a given instant t:

$$\begin{aligned}
[\![d]\!]_\sigma(t) &\stackrel{\text{def}}{=} d \\
[\![x(e)]\!]_\sigma(t) &\stackrel{\text{def}}{=} \begin{cases} \bot^D_{\text{outside}} & \text{if } [\![e]\!]_\sigma(t) = \bot^{\mathbb{T}}_{\text{outside}} \\ v & \text{if } [\![e]\!]_\sigma(t) = t' \text{ and } \sigma_x(t') = v \end{cases} \\
[\![\texttt{f}(E_1, \ldots, E_k)]\!]_\sigma(t) &\stackrel{\text{def}}{=} f([\![E_1]\!]_\sigma(t), \ldots, [\![E_k]\!]_\sigma(t)) \\
[\![\texttt{t}]\!]_\sigma(t) &\stackrel{\text{def}}{=} t \\
[\![\tau_x]\!]_\sigma(t) &\stackrel{\text{def}}{=} [\![\tau_x]\!]_\sigma(t) \\
[\![\texttt{outside}_D]\!]_\sigma(t) &\stackrel{\text{def}}{=} \bot^D_{\text{outside}} \\
[\![\texttt{notick}_D]\!]_\sigma(t) &\stackrel{\text{def}}{=} \bot^D_{\text{notick}}
\end{aligned}$$

Note that $[\![x(e)]\!]_\sigma$ includes the possibility that (1) the expression cannot be evaluated because the time instant given by $[\![e]\!]_\sigma(t)$ is outside the boundaries of domain of the stream and (2) the expression is not defined because the stream does not tick at t. It is easy to see that the cases for $[\![x(e)]\!]_\sigma$ are exhaustive because $[\![e]\!]_\sigma(t)$ guarantees that $\sigma_x(t')$ is defined.

[3] we use colors to better distinguish between semantic maps

For example, consider the following stream $(1.0, 17), (2.5, 21), (3.5, 12)$ for variable `sale` from Example 1. Then

$$[\![\texttt{sale(~t)}]\!]_\sigma(3.1) = [\![\texttt{sale(sale<~ t)}]\!]_\sigma(3.1) = [\![\texttt{sale}]\!]_\sigma(2.5) = 21$$

Definition 2 (Evaluation Model). *Given a valuation σ of variables $I \cup O$ the evaluation of the equations for stream $y \in O$ is:*

$$[\![T_y, V_y]\!]_\sigma \stackrel{\text{def}}{=} \{(t, d) \mid t \in [\![T_y]\!]_\sigma \text{ and } d = [\![V_y]\!]_\sigma(t) \text{ and } d \neq \bot^D_{notick}\}$$

An evaluation model is a valuation σ such that for every $y \in O$: $\sigma_y = [\![T_y, V_y]\!]_\sigma$.

The goal of a Striver specification is to define a monitor, that intuitively should be a computable function from input streams into output streams. The following definition captures whether a specification indeed corresponds to such a function.

Definition 3 (Well-defined). *A specification φ is* well-defined *if for all σ_I, there is a unique σ_O, such that $\sigma_I \cup \sigma_O$ is an evaluation model of φ.*

As with synchronous SRV, specifications can be ill-defined. For example, the following specification (`define bool a:= not a`) admits no evaluation model, and (`define bool a:= a`) admits many evaluation models. Additionally, a specification is efficiently monitorable if the output at time t only depends on the input at time t, which enable the incremental computation of the output stream.

Definition 4 (Efficiently monitorable). *A well-defined specification φ is* efficiently monitorable *whenever for every two input σ_I and σ'_I with evaluation models σ_O and σ'_O, and for every time t, if $\sigma_I|_t = \sigma'_I|_t$ then $\sigma_O|_t = \sigma'_O|_t$.*

2.4 Well-formedness

The condition of well-definedness is a semantic condition, which is not easy to check for a given specification (undecidable for expressive enough domains). We present here a syntactic condition, called well-formedness, that is easy to check on input specifications and guarantees that specifications are well-defined. Most specifications encountered in practice are well-formed.

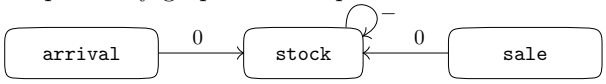
We first define a subset of the offset expressions, called the *Present* subset, as the smallest subset that contains `t` and such that if $e \in \textit{Present}$ then $(x \texttt{<~} e) \in \textit{Present}$. We say that an output stream variable y directly depends on a stream variable x (and we write $x \to y$) if x appears in T_y or V_y. We say that y has a present direct dependency on x (and write $x \xrightarrow{0} y$) if $x \to y$ and either

- $x.\textit{ticks}$ appears in T_y, or
- $(x\texttt{<~} e)$ appears in V_y and $e \in \textit{Present}$.

A direct dependency captures whether in order to compute a value of a stream variable y at position t, it is necessary to know the value of stream variable x up to t. If $x \to y$ but $x \not\xrightarrow{0} y$ we say that y directly depends on x in the past (and we write $x \xrightarrow{-} y$).

Definition 5 (Dependency Graph). *The dependency graph of a specification φ is a graph (V, E) where $V = I \cup O$ and $E = V \times V \times \{\xrightarrow{0}, \xrightarrow{-}\}$.*

The dependency graph of Example 1 is:



The following definition captures whether an output stream variable cannot depend on itself at the present moment.

Definition 6 (Well-Formed Specifications). *A specification φ is well-formed if every closed path in its dependency graph contains a past dependency edge $\xrightarrow{-}$.*

Closed paths in the dependency graph correspond to dependencies between a stream and itself in the specification φ. These closed paths do not create problems if the path corresponds to accessing the strict past of the stream. Note that if one removes $\xrightarrow{-}$ edges from the dependency graph of a well-formed specification, the resulting graph is necessarily a DAG. In other words $\xrightarrow{0}{}^*$ is irreflexive. The following lemma formally captures the information that is sufficient to determine the value of a given stream at a given time instant.

Lemma 1. *Let y be an output stream variable of a specification φ, σ, σ' be two evaluation models of φ, such that, for time instant t:*
(i) For every variable x, $\sigma_x(t') = \sigma'_x(t')$ for every $t' < t$, and
(ii) For every x, such that $x \xrightarrow{0}{}^* y$, $\sigma_x(t') = \sigma'_x(t')$ for every $t' \leq t$
Then $\sigma_y(t) = \sigma'_y(t)$.

The proof proceeds by structural induction on expressions, with the observation that only values in the past are necessary, as in conditions (i) and (ii). We are now ready to show that well-formed specifications cannot have two different evaluation models.

Theorem 1. *Every well-formed Striver specification is well-defined.*

The proof proceeds by showing that for well-formed specifications two evaluation models must be equal. This is shown by induction on the events in the traces to prove that the i-th event must be identical. Lemma 1 guarantees that induction can be applied.

3 Operational semantics

The semantics of Striver specifications introduced in the previous section are denotational in the sense that these semantics associate for a given input stream valuation exactly one output stream valuation, but does not provide a procedure to compute the output streams, let alone do it incrementally. We provide in this section an operational semantics that computes the output incrementally . We fix a specification φ with dependency graph G and we let $G^{=}$ be its pruned

dependency graph (obtained from G by removing $\xrightarrow{0}$ edges). We also fix $<$ to be an arbitrary total order between stream variables that is a reverse topological order of $G^{=}$.

We first present an online monitoring algorithm that stores the full history computed so far for every output stream variable. Later we will provide bounds on the portion of the history that needs to be remembered by the monitor, showing that only a bounded number of events needs to be recorded, and that this bound depends only on the size of the specification (number of streams) and not on the length of trace. This modified algorithm is a trace-length independent monitor for efficiently monitorable Striver specifications. The algorithm maintains the following state (H, t_q):

- **History**: H is a finite event stream one for each output stream variable. We use H_y for the event stream prefix for stream variable y.
- **Quiescence time**: t_q is the time up to which all output streams have been computed.

The monitor runs a main loop, calculating first the next relevant time t_q for the monitoring evaluation and then computing all outputs (if any) for time t_q. We show that no event exists in any stream in the interval between two consecutive quiescence time instants. We assume that at time t, the next event for every input stream is available to the monitor, even though knowing that there is no event up-to some t_q is sufficient.

The core observation follows from Lemma 1, which limits the information that is necessary to compute whether stream y at instant t contains an event (t, d). All this information is contained in H, so we write $[\![T_y]\!]_H$ and $[\![V_y]\!]_H$ to remark that only H is needed to compute $[\![T_y]\!]_\sigma$ and $[\![V_y]\!]_\sigma$.

The main algorithm, Monitor, is shown in Algorithm 1. Lines 2 and 3 set the history and initial quiescence time. The main loop continues until no more events can be generated. Line 5 computes the next quiescence time, by taking the minimum instant after the last quiescence time at which some output stream may tick. A stream y "votes" (see Algorithm 2) for the next possible instant at which its ticking equation T_y can possibly contain a value. Consequently, if no input stream votes for an earlier time it is guaranteed that no ticking equation will contain a value t lower than the lowest vote. Note that recursive calls at line 28 terminate because the graph $G^{=}$ is acyclic (recall that the specification is well-formed).

The algorithm follows a topological order over the $G^{=}$, so the information about the past required in Lemma 1 is contained in H. The following result shows that, assuming that σ_I is non-zeno, the output is also non-zeno. Hence, for every instant t, the algorithm eventually reaches $t_q > t$ in a finite number of executions of the main loop.

Lemma 2. Monitor *generates non-zeno output for a given non-zeno input.*

The proof proceeds by contradiction assuming a t with non-zeno output, and the minimum output stream in $G^{=}$ that has a non-zeno output, and then showing

Algorithm 1 MONITOR: Online Monitor

```
1: procedure MONITOR
2:     H_s ← ⟨⟩ for every s
3:     t_q ← −∞
4:     loop                                                        ▷ Step
5:         t_q ← min_{s∈O} {t | t = VOTE(H, T_s, t_q)}
6:         if t_q = ∞ then break
7:         for s in G^= following < do
8:             if t_q ∈ [[T_s]]_H then
9:                 v ← [[V_s]]_H(t_q)
10:                if v ≠ ⊥^D_notick then
11:                    H_s ← H_s ++ (t_q, v)                       ▷ Updates history H
12:                    emit(t_q, v, s)
13:        end for
14:    end loop
```

Algorithm 2 VOTE: Compute the next ticking instant

```
15: function VOTE(H, expr, t)
16:     switch expr do
17:         case delay(s)
18:             if (t' + v) > t (where (t', v) = last(H_s)) then return t' + v
19:             else return ∞
20:         case {c}
21:             if c > t then return c
22:             else return ∞
23:         case a ∪ b
24:             return min(VOTE(H, a, t), VOTE(H, b, t))
25:         case y.ticks with y ∈ O
26:             return VOTE(H, T_y, t)
27:         case s.ticks with s ∈ I
28:             return succ_>(σ_s, t_q)
```

that there must be a non-zeno output for $t - \epsilon$. This can be applied $\frac{t}{\epsilon}$ times to conclude that there is non-zeno output before 0 which is a contradiction.

We finally show that the output of MONITOR is an evaluation model. We use $H_s^i(\sigma_I)$ for the history of events H_s after the i-th execution of the loop body, and $H_s^*(\sigma_I)$ for the sequence of events generated after a continuous execution of the monitor. Note that $H_s^*(\sigma_I)$ can be a finite sequence of events (if the input is bounded and no repetition is introduced in the specification using `delay`) or an infinite sequence of events. In the first case, the vote is eventually ∞ and the monitoring algorithm halts.

Theorem 2. *Let σ_I be an input event stream, and let σ_O consist of $\sigma_x = H_x^*(\sigma_I)$ for every output stream x. Then (σ_I, σ_O) is an evaluation model of φ.*

The proof proceeds by induction on the number of rounds in the loop, showing that the output is an evaluation model up-to the quiescence time. Putting together Theorem 2, Lemma 1 and Lemma 2 we obtain the following result.

Corollary 1. *Let φ be a well-formed specification, σ_I a non-zeno input stream and H^* the result of* Monitor*. Then, H^* is the only evaluation model for input σ_I, and H^* is non-zeno.*

Trace Length Independent Monitoring The algorithm Monitor shown above computes incrementally the only possible evaluation model for a given input stream, but this algorithm stores the whole prefix H_y for every output stream variable y. We show now a modification of the algorithm that is trace length independent, based on flattening the specification. A specification is *flat* if every occurrence of an offset expression in every T_y is either $x(\texttt{<\~{} t})$ or $x(\texttt{<< t})$. In other words, there can be no nested term of the form $x(\texttt{<\~{}}\ (y\texttt{<\~{}}\ \texttt{t}))$ or $x(\texttt{<\~{}}\ (y\texttt{<<}\ \texttt{t}))$ or $x(\texttt{<<}\ (y\texttt{<\~{}}\ \texttt{t}))$ or $x(\texttt{<<}\ (y\texttt{<<}\ \texttt{t}))$. We first show that every specification can be transformed into a flat specification. The flattening applies incrementally the following steps to every nested term $x(E(y\texttt{<<}\ \texttt{t}))$, where E is an arbitrary offset term:

1. introduce a fresh stream s with equations $T_s = y.ticks$ and $V_s = x(E(\texttt{t}))$
2. replace every occurrence of $x(E(y\ \texttt{<<}\ \texttt{t}))$ by $s\texttt{(<t)}$.

Example 3. Consider the following specification of a continuous integration process in software engineering. The intended meaning is to report in `faulty` those commits to a repository that fail the unit tests.

```
input commit_id commits, unit push, bool tests
ticks faulty             := tests.ticks
define commit_id faulty := if tests(~t) then notick
                                        else commits(<push<<t)
```

After applying the flattening process the specification becomes:

```
define commit_id faulty := if tests(~t) then notick else s(<t)
ticks              s := push.ticks
define  commit_id s := commits(<t)
```

Here, `s` stores the `commit_id` of the last commit at the point of a `push`, which is precisely the information to report at the time of a `faulty` commit. □

Lemma 3. *Let φ be a specification. There is an equivalent flat specification φ' that is linear in the size of φ.*

Now, let φ' be the flat specification obtained from φ and let y be an output stream variable. Consider the cases for offset sub-expressions in the computation of $[\![V_y]\!]_H(t)$ in line 9 of Monitor:

- $s\texttt{<\~{}}\ t$: the evaluation fetches the value H_s at time t (if s ticks at t) of at the previous ticking time (if s does not tick at t).
- $s\texttt{<<}\ t$: the evaluation fetches the value H_s at the previous ticking time of s.

In either case, only the last two elements of H_s are needed. The similar argument can be made to compute T_y because only the last event of s is needed for $\texttt{delay}(s)$. Hence, to evaluate MONITOR on flat specifications, the algorithm only needs to maintain the last two elements in the history for every output stream variable to compute the next value of every value and ticking equation.

Theorem 3. *Every flat specification φ can be monitored online with linear memory in the size of the specification and independently of the length of the trace. Moreover, every step can be computed in linear time on the size φ.*

4 Extensions and Comparison

We first sketch how to define the most complex operator[4] of TeSSLa: $x = \mathit{delay}\langle s_0, s_1\rangle$, which creates an event stream x whick will tick at an instant t if there is an event (t', v) in s_0 such that $t' + v = t$ and also $\mathit{dom}(s_1) \cap (t', t) = \emptyset$ TeSSLa does not handle explicit time and offsets but builds specifications from building blocks like *delay*. Given inputs `s0` and `s1` the Striver specification is:

```
ticks aux := s0.ticks U s1.ticks
define Time_eps aux := if isticking(s1) then infty
    else if aux(<t,infty) = infty || aux(<t) + aux<<t <= t
         then s0(~t) else notick
ticks x := delay x_aux
define unit x := ()
```

We now present three extensions to the basic Striver introduced previously.

Accessing successors. The first extension allows accessing future events, via the dual of the offset operators x `>~` e and x `>>` e, and the syntactic sugar to access the successor value $x\texttt{(}e\texttt{>)}$, $x\texttt{(}e\texttt{~)}$, $x\texttt{(}e, d\texttt{>)}$ and $x\texttt{(}e, d\texttt{~)}$. As for Lola, well-formedness can be guaranteed as long as all strongly connected components in the dependency graph contain only $\xrightarrow{-}$ and $\xrightarrow{0}$ edges, or only $\xrightarrow{+}$ and $\xrightarrow{0}$ edges, and additionally, there is no cycle with only $\xrightarrow{0}$ edges. For example, this guarantees that there is no cyclic dependency, as every stream either depends on itself in the future or in the past (or none at all).

All Delays. This allows defining tick sets that consider all delays. The ticking expressions are extended with an operator `delayall` with the following semantics:

$$[\![\texttt{delayall}(w)]\!]_\sigma \stackrel{\text{def}}{=} \{t' \mid \text{ there is a } t \in \mathit{dom}(\sigma_w) \text{ such that } t + \sigma_w(t) = t'\}$$

This extension requires only to change VOTE to accommodate for a set of possible pending delays and not just a single delay. In general, this cannot be implemented in finite memory for arbitrary event rates and delays, but MONITOR works directly for the online monitoring this construct.

[4] Due to limitations a full comparison against TeSSLa and STL is not presented here.

Windows. The last extension allows implementing computations over precise windows, like "*count the number of events in every window of one second*". This cannot be described in TeSSLa [9], which is limited to finite memory monitors, or in RTLola [14] because this specification is not isochronous. Note that this property cannot be monitored by splitting the time in intervals of one second and counting the events in each of the intervals obtained (as in RTLola) as this approach misses the case of counting the events in part of one window and the remaining time in the adjacent window. The main idea of this extension is to enrich time expressions with a tag, in such a way that every tick carries an additional value (we called this extension *dependent time*). Then, `delay` and `delayall` are enriched with the ability to use tagged time streams, with the caveat that the `U` combinator must now indicate how to combine tags. Consider the following example with input `int s`:

```
ticks wcount :=  (const 1 s)  U delayall (const (-1,5) s)
define int wcount t aux := wcount(<t,0) + aux
```

The stream `wcount` must only be computed when a new event arrives in s (adding 1) or when an event leaves the window (substracting 1), which is monitored with a constant number of operations per event, but requires storing a number of events that depends on the event rate.

The Signal Temporal Logic STL [21,1]—when interpreted over piecewise-constant signals—is subsumed by Striver. First note that event streams have a dual interpretation as piece-wise constant signals, where the signal only changes at the point where events are produced. The translation to Striver opens the door to a quantitative computation of STL by enriching the data types of expressions and verdicts. We show the operator $x\,\mathcal{U}_{[0,b]}\,y$:

```
ticks v := x U y U delayall -b x U delayall -b y
define bool v t := if  y(~t,false) then true else
    if !x(~t,false) then false else
    let t' := yT(t~) in
    if t'==outside || t' > t+b then false else t' <= xF(t~)
```

5 Empirical Evaluation

We report an empirical evaluation of a prototype sequential Striver implementation, written in the Go programming language[5]. We measure the memory usage and time per event for two collections of specifications. The first collection, from Example 1, computes the stocks of p independent products. These specifications contain a number of streams proportional to p, where each defining equation is of constant size. The second collection computes the average of the last k sales of a fixed product, via streams that tick at the selling instants and compute the sum of the last k sales (see the appendix for the concrete specs). The resulting specifications has depth proportional to k. We instantiate k and p from 10 to 500 and run each resulting specification with a set of generated input traces. We run the experiments on a virtual machine on top of an Intel Xeon at 3GHz with

[5] Striver is available at `http://github.com/imdea-software/striver`

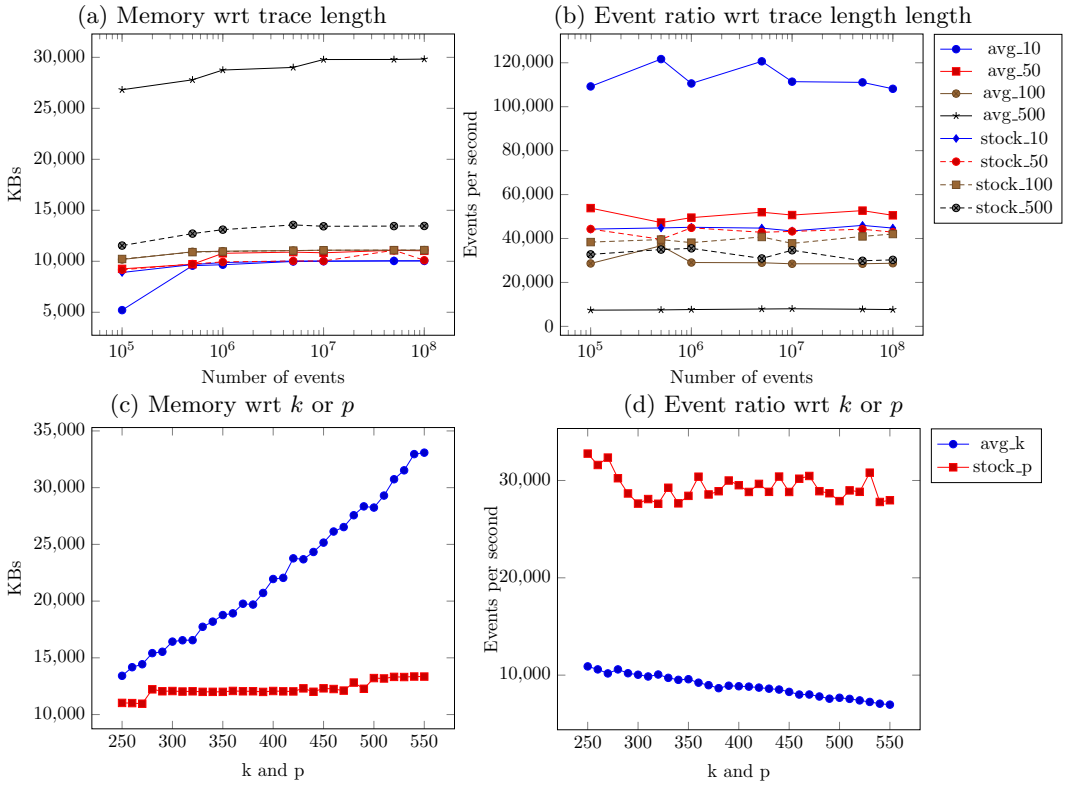


Fig. 1. Empirical evaluation

32GB of RAM, and measure the average memory usage (using the OS) and the number of events processed per second.

In the first experiment, we run the synthesized monitors with traces of varying length (top two plots in Figure 1). The results illustrate that the memory needed to monitor each specification is independent of the length of the trace (the curves are roughly constant). Also, the ratio of events processed is independent of the length of the trace. In the second experiment, we fix a trace of 1 million events and run the specifications with k and p ranging from 250 to 550. The results (lower diagrams) indicate that the memory needed to monitor `stock_p` is independent of the number of products while the memory needed to monitor each `avg_k` specification grows linearly with k. Recall that theoretically all specifications can be monitored with memory linearly on the size of the specification.

6 Conclusion and Future Work

We have introduced Striver, a specification language with explicit time and offset reference for the stream runtime verification of timed event streams. We have presented a trace-length independent online monitoring algorithm for the efficiently monitorable fragment. Future work includes the extension of the language with parametrization, (like in QEA [4], MFOTL [6] and Lola2.0 [13]), to dynamically instantiate monitors for observed data items. We are also studying offline evaluation algorithms, and algorithms that tolerate deviations in the time-stamps and asynchronous arrival of events from the different input streams.

References

1. *Lectures on Runtime Verification*, volume 10457 of *LNCS*, chapter Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications, pages 135–175. Springer, 2018.
2. *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*. Springer, 2018.
3. Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *J. ACM*, 49(2):172–206, 2002.
4. Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In *Proc of the 18th Int'l Symp. on Formal Methods (FM'12)*, volume 7436 of *LNCS*, pages 68–84. Springer, 2012.
5. Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *Proc. of VMCAI'04*, LNCS 2937, pages 44–57. Springer, 2004.
6. David A. Basin, Felix Klaedtke, Samuel Mller, and Eugen Zalinescu. Monitoring metric first-order temporal properties. *Journal of the ACM*, 62(2), 2015.
7. Andreas Bauer, Martin Leucker, and Chrisitan Schallhart. Runtime verification for LTL and TLTL. *ACM T. Softw. Eng. Meth.*, 20(4):14, 2011.
8. Laura Bozelli and César Sánchez. Foundations of Boolean stream runtime verification. In *In Proc. RV'14*, volume 8734 of *LNCS*, pages 64–79. Springer, 2014.
9. Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. TeSSLa: Temporal stream-based specification language. In *Proc. of the 21st. Brazilian Symp. on Formal Methods (SBMF'18)*, volume 11254 of *LNCS*. Springer, 2018.
10. Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.
11. Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. LOLA: runtime monitoring of synchronous systems. In *Proc. of TIME'05*, pages 166–174. IEEE, 2005.
12. Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In *Proc. of CAV'03*, volume 2725 of *LNCS 2725*, pages 27–39. Springer, 2003.
13. Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. A stream-based specification language for network monitoring. In *Proc. of the 16th Int'l Conf. on Runtime Verification (RV'16)*, volume 10012 of *LNCS*, pages 152–168. Springer, 2016.
14. Peter Faymonville, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. Real-time stream-based monitoring. *CoRR*, abs/1711.03829, 2017.
15. Alwyn E. Goodloe and Lee Pike. Monitoring distributed real-time systems: A survey and future directions. Technical report, NASA Langley Research Center, 2010.
16. Klaus Havelund and Allen Goldberg. Verify your runs. In *Proc. of VSTTE'05*, LNCS 4171, pages 374–383. Springer, 2005.
17. Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In *Proc. of TACAS'02*, LNCS 2280, pages 342–356. Springer, 2002.

18. Martin Leucker. Teaching runtime verification. In *Proc. of RV'11*, number 7186 in LNCS, pages 34–48. Springer, 2011.
19. Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm. TeSSLa: Runtime verification of non-synchronized real-time streams. In *Proc. of the 33rd Symposium on Applied Computing (SAC'18)*. ACM, 2018.
20. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Logic Algebr. Progr.*, 78(5):293–303, 2009.
21. Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *Proc. of FORMATS/FTRTFT 2004*, volume 3253 of *LNCS*, pages 152–166. Springer, 2004.
22. Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A hard real-time runtime monitor. In *Proc. of RV'10*, LNCS 6418. Springer, 2010.
23. Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. In *Proc. of FM'06*, LNCS 4085, pages 573–586. Springer, 2006.
24. Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 12(2):151–197, 2005.
25. Koushik Sen and Grigore Roşu. Generating optimal monitors for extended regular expressions. *ENTCS*, 89(2):226–245, 2003.

A Missing Proofs

Lemma 1. *Let y be an output stream variable of a specification φ, σ, σ' be two evaluation models of φ, such that, for time instant t:*

(i) For every variable x, $\sigma_x(t') = \sigma'_x(t')$ for every $t' < t$, and
(ii) For every x, such that $x \xrightarrow{0}{}^{*} y$, $\sigma_x(t') = \sigma'_x(t')$ for every $t' \leq t$
Then $\sigma_y(t) = \sigma'_y(t)$.

Proof. Note that since σ_y and σ'_y must satisfy that $T_y = [\![T_y]\!]_\sigma$ and $T_y = [\![T_y]\!]_{\sigma'}$, and also $V_y[\![V_y]\!]_\sigma$ and $V_y = [\![V_y]\!]_{\sigma'}$. It is easy to see that $t \in [\![T_y]\!]_\sigma$ if and only if $\in [\![T_y]\!]_{\sigma'}$, by structural induction on ticking expressions. The key observation is that only values in the conditions of the lemma are needed for the evaluation, which are assumed to be the same in σ and σ'. Similarly, it is easy to see that $[\![V_y]\!]_\sigma = [\![V_y]\!]_{\sigma'}$ because again the values needed are the same in σ and σ'. □

Theorem 4. *Every well-formed Striver specification is well defined.*

Proof. Let φ be a well-formed specification and σ and σ' two evaluation models for the same input (that is $\sigma_I = \sigma'_I$). We show that $\sigma = \sigma'$. Since φ is well-formed, the dependency graph removing $\xrightarrow{-}$ is acyclic. Let $<$ be an arbitrary total order between stream variables such that if $x \xrightarrow{0}{}^{*} y$ then $x < y$. Note that $<$ is simply a reverse topological order if this acyclic graph. Now we derive a total order between the events occurring in streams in σ as follows. Let $(t, d) \in \sigma_x$ and $(t', d)' \in \sigma_y$ be two such events. Then

- $(t, d) \prec (t', d')$ whenever $t < t'$, or
- $(t, d) \prec (t', d')$ whenever $t = t'$ and $x < y$.

Since $<$ is a total order between variables, $\prec$ is also a total order between all events in σ. We now show by induction in the total order that $\sigma = \sigma'$. Consider the first event (t, d) in σ according to $\prec$. This event can be either:

- an input event. In this case, since $\sigma_I = \sigma'_I$, (t, d) must also be the first input event in σ'_I. The only possibility for σ' to differ in the first event is that some output stream y has an event $(t'd') \prec (t, d)$. But this is only possible if the defining equations for y with no previous events make $t' \in [\![T_y]\!]_\sigma$ and $d' = [\![V_y]\!]_\sigma$, but by Lemma 1 (t', d) is also an event in σ_y, which contradicts that (t, d) was the first event.
- an event in an output stream y. In this case, again $t \in [\![T_y]\!]_\sigma$ if and only if $t \in [\![T_y]\!]_{\sigma'}$ and $d = [\![V_y]\!]_\sigma = [\![V_y]\!]_{\sigma'}$.

Assume now the inductive hypothesis that both streams coincide up to the i-th event and consider the i+1 event in σ. Again, if the event (t, d) is an input event it must also by the following input event in σ', and no output event can precede it because it would also precede (t, d) in σ because the defining equations depend on the i-th preceding events. Also, if the event (t, d) is an output events, since the evaluations of the defining events are the same in σ and σ', (t, d) will also be an event in σ' (and cannot be preceded by another event). This finishes the proof. □

Lemma 2. MONITOR *generates non-zeno output for a given non-zeno input.*

Proof. Note that events are generated in strictly increasing time for every stream, because the quiescent time t_q decided in line 5 is greater than the current time. However, that does not imply non-zenoness because some time domains (like the reals and the rationals) accept infinite sequences of increasing time stamps do not pass a given instant t.

Now, we first show that if the output generated by the monitor is zeno for time t (that is, there is no bound on the executions of the loop body that make $t_q > t$) then the execution is also zeno for time $t - \epsilon$. The lemma then follows because, by repeating the result $\frac{t}{\epsilon}$ times we will obtain that there is a zeno execution that does not pass $t - \epsilon\frac{t}{\epsilon} = 0$, but the second execution already passes 0.

Consider one such offending t. There must be an output stream variable x that votes infinitely many times in the infinite sequence of increasing quiescence times that never pass t. Let x be the lowest such stream variable in $(G^{=}, <)$. Consider the ticking expression for x. Since $\cup$ collects the votes for its sub-expressions, it follows that some sub-expression votes for infinitely many quiescent times in the sequence. The sub-expression cannot be $s.ticks$, because s would be lower than x in $<$ (contracting that x is minimal). Hence, the sub-expression voting infinitely many times is of the form $\texttt{delay}(s)$. Then, all these votes are caused by different events in H_s that are ticks of s that happened earlier than $t - \epsilon$. □

Theorem 2. *Let σ_I be an input event stream, and let σ_O consist of $\sigma_x = H^*_x(\sigma_I)$ for every output stream x. Then (σ_I, σ_O) is an evaluation model of φ.*

Proof. Let σ be (σ_I, σ_O). By Lemma 2 the sequence of quiescent times is a non-zeno sequence. We show by induction on the votes of MONITOR that for every quiescent time t_q, σ is an evaluation model up-to t_q, that is $H^*_x|_{t_q} = [\![T_x, V_x]\!]_\sigma|_{t_q}$.

Consider a quiescent time t_q^{prev} and let $t_y = \text{VOTE}(H, y.ticks, t_q^{\text{prev}})$. We first show that for every output stream y, $t_y \in [\![T_y]\!]_\sigma$ and for no t' with $t_q^{\text{prev}} < t' < t_y$, $t' \in [\![T_y]\!]_\sigma$. This results follows by induction on $<$, by Lemma 1 which guarantees that only the past is necessary to evaluate $[\![T_y]\!]_\sigma$, and by our assumption that σ is an evaluation model up-to t_q^{prev}. Now, let t_q be the next quiescence time after t_q^{prev} chosen in line 5. We show, again by induction on $<$, that for every output stream variable y, H_y contains an event (t_q, v) if and only if $t_q \in [\![T_y]\!]_\sigma$ (which we showed above), and $v = [\![V_y]\!]_\sigma = [\![V_y]\!]_H$ as computed in line 9. Hence, all events in H_y satisfy that $(t_q, v) \in [\![T_y, V_y]\!]_\sigma$ and all events $(t_q, v) \in [\![T_y, V_y]\!]_\sigma$ are added to H_y at quiescence time t_q. Since only quiescent times can satisfy $[\![T_y]\!]_\sigma$, it follows that σ is an evaluation model up-to t_q if σ is an evaluation model up-to t_q^{prev}, as desired. Finally, since the set of quiescent times is non-zeno, for every t there is a finite number n of executions of loop body after which $t_q^n \geq t$. Then, after n rounds σ is guaranteed to be an evaluation model up-to t. Since t is arbitrary, it follows that σ is an evaluation model. □

Consider two specifications $\varphi(I, O)$ and $\varphi'(I, O')$ with the same set of input stream variables, and with $O \subseteq O'$. We say that φ and φ' are equivalent whenever

for every σ_I, the valuation models σ_O of φ and σ_O of φ' coincide in O. The following lemma expresses that the flattening construction preserves equivalence.

Lemma 1. *Consider a specification φ and let φ' be a resulting specification obtained by applying the transformation once. Then φ and φ' are equivalent.*

B More examples

The following example illustrates a more complex specification.

Example 4. The following defines a *Bool* stream to monitor whether the stock falls below a predefined `threshold`.

```
const threshold := 100
ticks low := stock.ticks
define bool low := stock(~t) < threshold
```

We can use the stream variable `low_stock` to inform the user that the stock is low. Note that `low_stock` will compute a value every time the stock changes, which could lead to too many alarms until the order is performed. To overcome this issue, one can define a stream which only reports the changing points of `low_stock` (the let clauses allow to cleanly define local values):

```
ticks new_low := low_stock.ticks
define bool new_low := let val := low(~t) in
                       let prev := low(<t) in
                 if val != prev then val else notick
```

Finally, the following specifications allows emitting an alarm (of type *unit*) every time the stock is low for a certain length of time `length`:

```
const length := 2h
output unit long_low_stock
output T alarm

ticks alarm := new_low.ticks
define T alarm := if !new_low(~t) then infty
                  else length

ticks long_low_stock := at alarm
define unit long_low_stock := ()
```

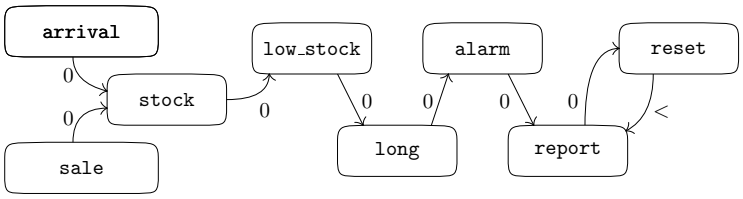
We can define a stream variable to report when there is a long period of low stock as defined by *long_low_stock*, but also to report if after a certain amount of time `report_time` the stock is ok using the following specification:

```
const report_time := 8h
output T clock_reset
output bool report

ticks clock_reset := report.ticks U {0}
define T clock_reset := report_time
```

```
ticks report := at clock_reset U long_low_stock.ticks
define bool report := isticking(long_low_stock)
```

The dependency graph of the given example is the following:



This finishes this example. □

Example 5. We also allow to apply a unary operation to a given stream $f(s)$, which would result in the creation of an intermediate stream with the same ticks as s but with $f(s(t))$ as the values. Given $s :: \mathcal{S}_A$ and a function $f :: A \to B$:

```
output B s_f_aux
ticks s_f_aux := s.ticks
define B s_f_aux :=
    f(s(~t))
```

Also, we ease the definition of delayed streams with a constant length with a similar trick. Given $s :: \mathcal{S}_\_$ and $length :: \mathbb{T}_\epsilon$:

```
output T_eps s_del_aux
ticks s_del_aux := s.ticks
define T_eps s_del_aux :=
    length
```

□

Example 6. The following specification adds the value of the event stream s within a window of 5 seconds:

```
ticks  int waggr :=  s.ticks  U delayall (pair 5sec -s)
define int waggr t aux := waggr(<<t,0) + aux
```

Example 7. Using this new extension, we can check STL properties by translating them to Striver specifications. To do so, we define a function $translate(\varphi, x, v)$ which will return a Striver specification with an output signal v and input signal x such that $v(t) = (x, t) \vDash \varphi$. The specification is constructed recursively over φ:

$translate(T, , v)$:

```
output bool v
ticks v := {0}
define bool v t :=
    true
```

$translate(\mu_f, x, v)$:

```
input D x
output bool v

ticks v := x.ticks
define bool v t :=
    f(x_1(~t),...,x_n(~t)) > 0
```

$translate(\neg\varphi, x, v)$, where PHISPEC = $translate(\varphi, x, vphi)$:

```
PHISPEC

output bool v

ticks v := vphi.ticks
define bool v t :=
    !vphi(~t)
```

$translate(\varphi \wedge \psi, x, v)$, where PHISPEC = $translate(\varphi, x, vphi)$ and PSISPEC = $translate(\psi, x, vpsi)$

```
PHISPEC

PSISPEC

output bool v

ticks v := vphi.ticks U vpsi.ticks
define bool v t :=
    vphi(~t) && vpsi(~t)
```

$translate(\varphi U_{[a,b]} \psi, x, v)$, where PHISPEC = $translate(\varphi, x, vphi)$ and PSISPEC = $translate(\psi, x, vpsi)$

```
PHISPEC

PSISPEC

output bool v
output bool phiF := filter (\v -> !v) vphi
output bool psiT := filter (\v -> v) vpsi

ticks v := delay -a vphi U delay -b vphi U delay -a vpsi U delay -b vpsi
define bool v t :=
    if psiT<~t >= t+a then true else
    if vpsi(~t) then true else
    let psiNextT := psiT~>t in
    if psiNextT == outside || psiNextT > t+b then false else
    if phiF~>t <= psiNextT then false else
    vphi(~t)
```

C TeSSLa operators

The equivalent Striver specifications of TeSSLa operators are defined as follows:

– **nil:** the stream $x = nil$ is defined as:

```
ticks x := {0}
define void x t := notick
```

– **unit:** the stream $x = unit$ is defined as:

```
ticks x := {0}
define unit x t := ()
```

– **lift**: the stream $x = lift\langle f, s_0, \ldots, s_n\rangle$:

```
input A0 s0
...
input An sn

ticks x := s0.ticks U ... U sn.ticks
define B x t :=
    if (s0<~t == outside || ... || sn<~t == outside)
    then notick else f(s0(~t),...,sn(~t))
```

– **time:** the stream $x = time\langle s\rangle$ is defined as:

```
input A s

ticks x := s.ticks
define Time x t := t
```

– **last**: the stream $x = last\langle s_0, s_1\rangle$:

```
input A0 s0
input A1 s1

ticks x := s1.ticks
define A0 x t := s0<<t
```

– **delay:** the stream $x = delay\langle s_0, s_1\rangle$ is defined as:

```
input Time_eps s0
input A s1
ticks x_aux := s0.ticks U s1.ticks
define Time_eps x_aux := if isticking(s1) then infty
    else if x_aux(<t,infty) == infty || x_aux(<t) + x_aux<<t <= t
         then s0(~t) else notick
ticks x := at x_aux
define unit x t := ()
```

Example 8. We consider the specification that allows to count events in a given input stream, which is a built-in block in TeSSLa (or a recursive definition in TeSSLa 2.0).

```
input A s

ticks event_count := s.ticks U {0}
define int x t :=
    if isticking(s) then event_count(<t,0) + 1 else 0
```

D Empirical Evaluation

The specification for the stock of p products is:

```
input int sale_1
input int arrival_1
...
input int sale_p
input int arrival_p

ticks stock_1 := sale_1.ticks U arrival_1.ticks
define int stock_1 := stock_1(<t,0) +
    (if isticking(arrival_1) then arrival_1(~t) else 0) -
    (if isticking(sale_1   ) then sale_1(~t   ) else 0)
...
ticks stock_p := sale_p.ticks U arrival_p.ticks
define int stock_p := stock_p(<t,0) +
    (if isticking(arrival_p) then arrival_p(~t) else 0) -
    (if isticking(sale_p   ) then sale_p(~t   ) else 0)
```

To compute the average of the last k purchases we use the following specification:

```
ticks denom := sale.ticks
define int denom := if denom(<t) == k
                    then k
                    else denom(<t,0)+1

ticks sumlastk := sale.ticks
define int sumlastk := sumlastk(<t,0) +
                       sale(~t) -
                       sale(<sale<<sale<<...<<t, 0)

ticks avgk := sale.ticks
define int avgk := sumlastk / denom
```