Noname manuscript No.
(will be inserted by the editor)

Stream runtime verification of real-time event-streams with the Striver Language

Felipe Gorostiaga[1,2,3] · **César Sánchez**[1]

the date of receipt and acceptance should be inserted later

Abstract In this paper we study the problem of runtime verification of real-time event streams, in particular we propose a language to describe monitors for real-time event streams that can manipulate data from rich domains.

We propose a solution based on stream runtime verification (SRV), where monitors are specified by describing how output streams of data are computed from input streams of data. SRV enables a clean separation between the temporal dependencies among incoming events, and the concrete operations that are performed during the monitoring.

Most SRV specification languages assume that all streams share a global synchronous clock, and divide time in discrete instants. At each instant every input has a reading and for every instant the monitor computes an output. In this paper we generalize the time assumption to cover real-time event streams, but keep the explicit time offsets present in some synchronous SRV languages like Lola. The language we introduce, called Striver, shares with SRV the simplicity and economy of operators, and the separation between the reasoning about time and the computation of data values. The version of Striver in this paper allows expressing future and past dependencies. Striver is a general language that allows expressing for certain time domains other real-time monitoring languages, like TeSSLa, and temporal logics, like STL.

We show in this paper translations from other formalisms for (piecewise-constant) real-time signals and timed event streams. Finally, we report an empirical evaluation of an implementation of Striver.

This work was funded in part by the Madrid Regional Government under project "S2018/TCS-4339 (BLOQUES-CM)", and by Spanish National Project "BOSCO (PGC2018-102210-B-100)".

Felipe Gorostiaga
E-mail: felipe.gorostiaga@imdea.org
César Sánchez
E-mail: cesar.sanchez@imdea.org

[1] IMDEA Software Institute, Madrid, Spain
[2] Universidad Politécnica de Madrid, Madrid, Spain
[3] CIFASIS, Argentina

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. Runtime verification is therefore a dynamic technique that considers only the traces observed, in contrast with static verification that must consider all executions of the system. Consequently, runtime verification sacrifices completeness to obtain a readily applicable formal method that can be combined with testing or debugging. Other common use of RV is the formal monitoring of reactive systems to provide explanations of the system behavior dynamically, which can be combined at runtime with corrective measures. See [1,2] for surveys on RV, and the recent book [3]. Early specification languages proposed in RV were based on temporal logics [4,5,6], regular expressions [7], timed regular expressions [8], rules [9], or rewriting [10]. The approach we propose here is based on stream runtime verification (SRV), pioneered by Lola [11].

Stream runtime verification defines monitors by declaring the dependencies between output streams (results) and input streams (observations from the system). The main idea of SRV is that the same sequence of steps performed during the monitoring of a temporal logic formula can be followed to compute statistics of

the input trace, as the temporal dependencies between the inputs and outputs are the same. The only necessary change is to generalize the operations performed from the data collected as input to compute the intermediate results and outputs, which allows computing richer verdicts, following the same steps and data dependencies. The generalization of the outcome of the monitoring process to richer verdict values brings runtime verification closer to monitoring and data stream-processing. See [12,13,14] for further works on SRV. Temporal testers [15], which can be seen as a Boolean SRV description, were later proposed as a similar monitoring technique for LTL. SRV was initially conceived for monitoring synchronous systems, where computation proceeds in cycles.

In the pioneering Lola SRV specification language, writing a specification consists of associating every output stream variable with a defining equation. The intention is that—once the input streams are known—each output variable is mapped to the unique output stream that satisfies its equation. Take, for example, the following Lola specification [11]:

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

This specification defines one intermediate stream, called `one_p`, and two output streams, called `always_p` and `count_p`. The output stream `always_p` captures whether the Boolean input stream `p` was true at every point in the past (that is, the LTL formula $\boxminus p$). The stream `one_p` is 1 when the input p is *true* and 0 otherwise, which eases the definition of `count_p` to count the number of times `p` was true in the past. Offset expressions like `count_p[-1,0]` or `always_p[-1,true]` refer to a different position in a stream with a default value when there is no such a position (that is, before the beginning and after then end of the trace). The offset expression `count_p[-1,0]` refers to stream `count_p` at the previous position with default value `0` if the referred position falls before the beginning of the trace. Similarly, the offset expression `always_p[-1,true]` refers to the previous position with default value `true`. 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 and next position at which some stream contains an event.

Other similar languages for timed event streams are TeSSLa [16] and RTLola [17] but both of these preclude to reason explicitly about real-time instants. Instead, TeSSLa and RTLola offer building blocks like stream transformers in the language to describe the temporal dependencies between streams. For this reason we say that Striver is an *explicit time* SRV formalism.

Striver is a stream-based declarative specification language for timed asynchronous observations, where streams are sequences of timed events. In other words, events in different streams do not necessarily happen at the same time. However, all time-stamps are totally ordered according to a global clock. This is the assumption made in the timed asynchronous model of distributed systems [18]. Striver targets the outline, non-intrusive monitoring of real-time systems. Outline in this context means that the monitor is not intertwined or modifies the system under analysis, but instead runs on its specific infrastructure, with the goal of minimizing the effect of monitoring on the system's behavior (non-intrusiveness). One of the most important concerns in RV is the usage of resources. The concept of *trace-length independence* refers to the ability of a monitoring algorithm to carry out the online evaluation (processing of input streams to generate output streams) with an amount of memory that can be bounded from the specification and is independent of the length of the trace.

Our intended application is the monitoring and testing of cloud systems and multi-core hardware monitoring, where our time assumption is reasonable. The Elastest project [19] aims at improving the testing of large cloud applications. The Elastest Monitoring Service (EMS) is a component of the Elastest infrastructure that improves the testing capabilities of Elastest. The core of the EMS is an implementation of the algorithms described in this paper.

Notions of time. The concepts of time used in this paper are summarized as follows:

- Synchronous SRV. Time proceeds as sequence of instants, where exactly one event is read in each input stream, and one output event is eventually computed for each output stream. Examples of specification languages that assume a synchronous model of time are Lola [11] and LTL [20].
- Isochronous SRV. The time domain is potentially continuous. Each observation is an event which carries a time-stamp, and can happen at any point. However, output streams are updated at periodic intervals (which justifies the name isochronous) or at those instants where there is an event in an input stream. Examples include the periodic stream definitions in RTLola [17].
- Asynchronous SRV. Again, the time domain is potentially continuous time, and time-stamped events can occur at any time. Output streams can contain

events at arbitrary points in time, without any period or input event. This requires the engine to generate events at arbitrary instants which justifies the name asynchronous. Examples are Striver [21] (extended in this paper) and TeSSLa [16]. RTLola [17] can generate events at periodic times and also at those times at which there is an input event.

Related work. TeSSLa [16] is a specification language for timed-event streams based on stream transformers (basic building blocks that take streams and define new streams). In contrast, Striver uses a style of specification that expresses the dependency of streams using explicit time offsets, in an approach more aligned to Lola. The seminal paper for TeSSLa [16] presents the language and [22] shows asynchronous operational semantics for a simpler fragment of the language (that encapsulates all recursion within the building blocks) that also disallows non-Zeno specifications. We prove in this paper that Striver subsumes TeSSLa (under some assumptions) in the sense that every stream transformer from TeSSLa can be implemented in Striver.

Another similar work is RTLola [17], which also aims to extend SRV from the synchronous domain to timed streams. In RTLola, defined streams are either computed at predefined periodic instants of time or at the ticking time of input streams. Even though the semantics of RTLola are given informally in [17], RTLola is either input driven or isochronous according to the definition above because output streams can only be generated at periodic times or at time triggered instants. RTLola is very efficient on inputs arriving at high speeds as a typical RTLola specification simply stores input events and computes output events (typically summaries) at regular intervals. However, this sacrifices trace length independence unless there is an assumption on the ratio of arrival of events. Compared to RTLola, in the model of computation of Striver, streams are computed at the specific real-time instants where they are required, resulting in a fully asynchronous SRV system. In this case, Striver is strictly more expressive than RTLola (the version from [17]) because RTLola cannot define properties that must be interpreted at every instant of time (like "there cannot be more than k events in any window of 3 s") which require to produce events in the output at instants that are neither periodic nor present in the input. It is simple to see that every construct in RTLola can be translated into a few lines of Striver code. Also, asynchronous languages like TeSSLa and Striver can be used more easily to define specifications that are guaranteed to be trace length independent, and be very efficient on inputs with sparse event but occasional heavy bursts.

Signal temporal logic (STL) [23,24] is a temporal logic for real-time signals based on metric temporal logic (MTL) [25] that is capable of dealing with numeric signals. We show in Sect. 5.2 that Striver can subsume STL over piecewise-constant signals and also generalize the semantics of STL to quantitative data collection over piecewise constant signal inputs.

Data Stream Management Systems (DSMS) [26] allow working with streams of input data by continuously executing queries over stored stream. Typically, DSMS queries are executed periodically and thus they present issues inherent to isochronous approaches. In particular, these systems are sensitive to sparse bursts of events, having to decide whether to buffer a rather large input data and keep the execution period high, or execute with a higher frequency, and waste CPU cycles when there is no data to consume. Also, the evaluation of queries typically comes in two flavors: (1) the ones that are evaluated over a fixed window of time, which may require only bounded resources but restrict the range of observations (for example in the Continuous Query Language (CQL) developed as part of the STREAM Data Stream Management System [27]); (2) the evaluations that store the whole history (which require unbounded storage). In comparison, one of the main concerns of stream runtime verification is to study rich monitoring languages with formal semantics that know the whole history of the computation and can be evaluated with bounded resources.

Contributions. In summary, the contributions of the paper are:

1. The Striver specification language, which generalizes (preserving the separation between data and time) SRV to timed event streams, keeping explicit time offsets, and not using additional building blocks or stream transformers.
2. A trace-length independent online algorithm for the past fragment, included in Sect. 4.1, and an online algorithm for the full language, included in Sect. 4.2 (which is not trace-length independent in general).
3. A comparison between TeSSLa and Striver, which is included in Sect. 5.1.
4. An extension of the language to describe sliding windows, in Sect. 5.2, which allows a translation from STL to Striver.
5. An empirical evaluation for both the past fragment and the extensions including future dependencies and windows, reported in Sect. 6.

Journal paper. An earlier version of this paper appears in the Proceedings of the 18th International Conference on Runtime Verification (RV'18) [21]. This paper

extends [21] including many proofs and additional examples, and more specifically, the following additions:

- An extension of Striver that includes future operators, which involves extending the syntax, type inference system, and generalizing the semantics and well-formedness condition.
- The complete proof of trace-length independence of the algorithm for the past fragment of Striver presented in [21] and a completely new online algorithm for the fully fledged version of Striver, included Sect. 4.2. This algorithm does not proceed synchronously as the simpler algorithm from but instead accesses each of the streams independently.
- The complete comparison with all operators of the TeSSLa specification language, included in Sect. 5.1.
- A further extension to define truly sliding windows in Striver (windows that span from any two points in the time domain) and the comparison with STL, included in Sect. 5.2. This requires the *bounded future fragment* of Striver, introduced in the same section.
- An extended empirical evaluation, in Sect. 6, particularly with new experiments to evaluate the extensions of the language.
- A discussion of the language properties in Sect. 7 including a finer-grain analysis on time boundaries, a sketch on how to perform offline monitoring which is a method to achieve trace length independence monitoring provided the appropriate host capabilities.

2 Preliminaries

The keystone of Stream Runtime Verification is to separate two concerns: the *temporal dependencies* and the *data* manipulated. The temporal dependencies are used to calculate the order of operations in monitoring algorithms, while the data manipulation describes how to perform each operation. We use here the term *data domains* to refer to the first-order signatures and structures that allows modeling the data manipulation. The clean separation between temporal dependencies and data domains allows generalizing existing algorithms that monitor temporal logics, from Boolean verdicts to quantitative verdicts, by using data domains richer than Booleans.

2.1 Data domains

We use many-sorted first order signatures and structures to describe data domains. A signature $\Sigma : \langle \mathcal{S}, \mathcal{F} \rangle$ consist on a finite collection of sorts $\mathcal{S}$, and function symbols $\mathcal{F}$ (where each argument of a function has a sort, and the resulting term also has a sort). A simple signature is *Booleans*, that has only one sort (*Bool*) with two constants (`true` and `false`), binary functions ($\wedge$, $\vee \dots$), unary functions like $\neg$, etc. In this paper, we use sort and type interchangeably. 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 $Nat \times Nat \to Nat$) as well as predicates $<$, $\leq$, which take two *Nat* arguments and return a *Bool* with their usual interpretation. We assume that all signatures have equality over all sorts and that every sort (*Nat*, *Bool*, *Queue*, *Stack*, etc.) is equipped with a ternary symbol $\texttt{if} \cdot \texttt{then} \cdot \texttt{else}\cdot$. In the case of *Nat*, the $\texttt{if} \cdot \texttt{then} \cdot \texttt{else}\cdot$ symbol has type $Bool \times Nat \times Nat \to Nat$.

The theories we consider are interpreted. Therefore, for every first-order signature there is a structure where all function symbols have a computational interpretation. That is, every 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 example, the symbol `+` can be used to construct an expression of type *Nat* given two expressions of type *Nat*, and `+` is associated with the interpreted function $+$ that computes the sum of two natural numbers. For simplicity, we omit the sort S from D_S (and simply write D) if it is clear from the context.

We will build specifications using *stream variables* to model input and output streams. Each stream variable is associated with a sort. From the point of view of syntactic expressions, stream variables are used to build atoms. As usual, given a set of sorted atoms A and a signature, the set of terms is the smallest set containing A and closed under the use of function symbols in the signature as term constructors (respecting sorts).

We consider a special *time* domain $\mathbb{T}$, whose interpretation is a (possibly infinite, possibly dense) totally ordered set. We also require the existence of a superset of the time domain $\mathbb{T}^+$ closed under addition $+$ (which is a total function), and such that the temporal domain $\mathbb{T}$ is an interval of $\mathbb{T}^+$. Usually, time domains contain a minimal element $\bar{0}$, a maximal element $\bar{1}$, or both, to denote the beginning and the end of time. Examples of time domains are $\mathbb{R}_{\geq 0}$, $\mathbb{Q}_{\geq 0}$, and $\mathbb{N}$, 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 $S \subseteq \mathbb{T}$ is non-Zeno when it does not contain bounded subsets with infinitely many elements, this is, whenever for every $t_a, t_b \in \mathbb{T}$, the set $S \cap [t_a, t_b]$ is finite.

We extend every domain D into $D_{\texttt{notick}}$ by including the fresh symbol $\bot_{\texttt{notick}}$ to indicate when a stream of type D does not contain an event. Additionally, we extend $D_{\texttt{notick}}$ into $D^{\bot}$ by including two additional fresh symbols: $\bot_{\texttt{-out}}$ and $\bot_{\texttt{+out}}$. We extend the equality function symbol in the signatures to deal with the introduced constants, where each constant is equal to itself, and different from the other constants and from all elements in the sorts of the underlying signature. The fresh symbols $\bot_{\texttt{-out}}$ and $\bot_{\texttt{+out}}$ are used to represent whether a time offset falls off the beginning or the end of the trace. We use $\mathbb{T}_{\texttt{-out}}$ for $\mathbb{T} \cup \{\bot_{\texttt{-out}}\}$, $\mathbb{T}_{\texttt{+out}}$ for $\mathbb{T} \cup \{\bot_{\texttt{+out}}\}$ and $\mathbb{T}_{\texttt{out}}$ for $\mathbb{T} \cup \{\bot_{\texttt{-out}}, \bot_{\texttt{+out}}\}$. Similarly, we use $D_{\texttt{-out}}$ for $D \cup \{\bot_{\texttt{-out}}\}$, $D_{\texttt{+out}}$ for $D \cup \{\bot_{\texttt{+out}}\}$ and $D_{\texttt{out}}$ for $D \cup \{\bot_{\texttt{-out}}, \bot_{\texttt{+out}}\}$.

A key principle in the design of Striver is that the implementation of data domains is used completely off-the-shelf, so the addition of these new symbols is performed within the Striver engine, and the actual off-the-self data domain implementation only receives actual values from the appropriate domains. Besides equality (which is easily extended as described above), we do not force any function in the theories (like + for example) to handle these new symbols.

2.2 Streams

Monitors observe sequences of events as inputs, where each event contains a data value from its domain and is time-stamped with an increasing time value. We model these sequences as event-streams. Given a partial function $f : A \rightharpoonup B$, we use $dom(f)$ as the subset of A where f is defined.

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.

The set $dom(\eta)$ is called the set of *event points* of η. An event stream η with a first element can be naturally represented as a *timed word*:

$$s_\eta = (t_0, \eta(t_0))(t_1, \eta(t_1)) \cdots \in (dom(\eta) \times D)^*,$$

such that:

1. s_η is ordered by time ($t_i < t_{i+1}$); and
2. the set $\{t \mid (t, d) \in s_\eta\}$ is non-Zeno.

Note that every sequence that is non-Zeno has first element if the time domain has minimum element (or if at least $dom(\eta)$ has a minimum element). If $dom(\eta)$ does not have a maximum element, we can extend time-words η into ω-timed words

$$s_\eta = (t_0, \eta(t_0))(t_1, \eta(t_1)) \cdots \in (dom(\eta) \times D)^\omega.$$

The set of all event streams over D is denoted by $\mathcal{E}_D$.

We introduce some additional notation for event streams to capture the previous and next event in the stream for a given point in time. 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 $prev_<$. Formally,

$$\begin{aligned} prev_<(\sigma, t) &\stackrel{\text{def}}{=} sup(dom(\sigma) \cap [\bar{0}, t)) \\ prev_\leq(\sigma, t) &\stackrel{\text{def}}{=} sup(dom(\sigma) \cap [\bar{0}, t]) \\ sup(S) &\stackrel{\text{def}}{=} \begin{cases} max(S) & \text{if } S \neq \emptyset \\ \bot_{\texttt{-out}} & \text{otherwise} \end{cases} \end{aligned}$$

The type of $prev_<$ and $prev_\leq$ is $\mathcal{E}_D \times \mathbb{T} \to \mathbb{T}_{\texttt{-out}}$. These functions can return $\bot_{\texttt{-out}}$ because sup returns $\bot_{\texttt{-out}}$ when the stream has no events in the interval provided. Note that $max(S)$ is well defined because time is totally ordered and every stream σ has a finite number of elements in every given interval. Similarly, given a stream σ and a time $t \in \mathbb{T}$, the expression $succ_>(\sigma, t)$ provides the nearest time instant in the future of t at which s is defined, and $succ_\geq(\sigma, t)$ returns t if $t \in dom(\sigma)$; otherwise, it behaves as $succ_>$. Formally,

$$\begin{aligned} succ_>(\sigma, t) &\stackrel{\text{def}}{=} inf(dom(\sigma) \cap (t, \bar{1}]) \\ succ_\geq(\sigma, t) &\stackrel{\text{def}}{=} inf(dom(\sigma) \cap [t, \bar{1}]) \\ inf(S) &\stackrel{\text{def}}{=} \begin{cases} min(S) & \text{if } S \neq \emptyset \\ \bot_{\texttt{+out}} & \text{otherwise} \end{cases} \end{aligned}$$

The type of $succ_>$ and $succ_\geq$ is $\mathcal{E}_D \times \mathbb{T} \to \mathbb{T}_{\texttt{+out}}$. These functions can return $\bot_{\texttt{+out}}$ because inf returns $\bot_{\texttt{+out}}$ when the stream has no event in the interval provided.

2.3 Efficient monitorability

A synchronous SRV specification that only refers to the past is called very efficiently monitorable (see [11]). In synchronous SRV, these specifications can be monitored online and guaranteed that (1) the online monitoring can be performed trace length independently (with an amount of memory that can be bounded a-priory and does not depend on the length of the trace), and (2) each output stream can be resolved immediately (that is, once all inputs are read at time t, all outputs for time t can be computed). The resources necessary to monitor a specification are considered relative to the size of data registers, meaning that for a trace length-independent specification, the engine requires a constant number of registers of the corresponding sort per stream. For example, trace length independence in logic requires a constant number of Boolean registers. In asynchronous

SRV formalisms like Striver, very efficiently monitorable specifications can be monitored preserving trace length independence. In the general case of unrestricted specifications the resources cannot be bounded at static time. We show in Sect. 4.1 an online monitoring algorithm for very efficiently monitorable Striver specifications and prove that this algorithm is trace length independent.

3 The Striver specification language

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*, which defines when the stream may contain an event;
> – a *value expression*, which defines the value contained in the event.

Note that in synchronous SRV, only a value expression is necessary because every stream has a value in every cycle (i.e., in every synchronous instant). Therefore, expressing explicitly when a stream produces a value in synchronous SRV would be redundant.

Formally, a Striver specification $\varphi : \langle I, O, V, C, T\rangle$ consists of input stream variables $I = \{x_1, \ldots, x_n\}$, output stream variables $O = \{y_1, \ldots, y_m\}$, a collection of clock or ticking expressions $C = \{C_1, \ldots, C_m\}$, a collection of value expressions $V = \{V_1, \ldots, V_m\}$, and a collection of sorts $T = \{T_1, \ldots, T_{n+m}\}$. Note that there is one ticking expression and one value expression per output stream variable, and one sort per stream variable. We assume $I \cap O = \emptyset$. We define the size[1] of a specification as its number of streams, this is, the size of the specification φ is $|I \cup O|$. Every output variable y is associated with a ticking expression $C_y \in C$ which captures when stream y *may* tick, and with a value expression $V_y \in V$ which defines what is the value of y when it ticks, and if it ticks at all. Every stream variable x is associated with a type name T_x that indicates its domain. Note that the sub-indices of C_y, V_y and T_x indicate the corresponding stream variable associated.

In practice, it is very useful that T_y defines an over-approximation of the set of instants at which y ticks. Then, the value expression can decide if the stream indeed produces a value or if the evaluation is a "no tick". A simple example of a filter can be seen in Example 1.

[1] a more refined version of size considers the size of the ticking and value expressions as well, but this is sufficient for the purpose of this paper.

3.1 Syntax

We fix Z to be the set of stream variables $Z = I \cup O$. There are three types of expressions: ticking expressions, value expressions and offset expressions. Offset expressions are used inside value expressions to allow temporal shifts. Formally, the expressions are:

– *Ticking Expressions*, which define when a stream may produce a value:

$$\alpha ::= \{c\} \mid v.\texttt{ticks} \mid \texttt{delay}\ \epsilon\ w \mid \alpha\ \mathsf{U}\ \alpha \qquad \text{(tick)}$$

where $c \in \mathbb{T}$, $\epsilon \in \mathbb{T}^+$ are constants (with $\epsilon \neq 0$), $v \in Z$ is an arbitrary stream variable, and U is used for the union of sets of ticks.

– *Offset Expressions*, which allow to fetch events from streams:

$$\tau_x ::= x\texttt{<~}\tau \mid x\texttt{<<}\tau \mid x\texttt{>~}\tau \mid x\texttt{>>}\tau$$
$$\tau ::= \texttt{t} \mid \tau_z \text{ for } z \in Z \qquad \text{(offset)}$$

The expression `t` represents the current time instant. The expression $x\texttt{<<}\tau$ is used to refer to the previous instant at which x ticks strictly in the past of τ (or $\bot_{\texttt{-out}}$ if there is not such an instant). The expression $x\texttt{<~}\tau$ also considers the present as a candidate instant. Analogously, the intended meaning of $x\texttt{>>}\tau$ is to refer to the next instant strictly in the future of τ at which x ticks (or $\bot_{\texttt{+out}}$ if there is not such an instant). The expression $x\texttt{>~}\tau$ also considers the present as a candidate.

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

$$E ::= d \mid x(\tau_x) \mid \texttt{f}(E_1, \ldots, E_k) \mid \tau$$
$$\mid \texttt{-out} \mid \texttt{+out} \mid \texttt{notick} \qquad \text{(value)}$$

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 if the point is within the time boundaries. Expressions `t` and τ_x build expressions of sort $\mathbb{T}_{\texttt{out}}$. The three additional constants `-out`, `+out` and `notick` allow reasoning (using equality) about accessing both ends of the streams, or not generating an event at a ticking candidate instant.

We also use the following syntactic sugar:

$$x\texttt{(~}e\texttt{)} \stackrel{\text{def}}{=} x(x\texttt{<~}e) \quad x\texttt{(<}e\texttt{)} \stackrel{\text{def}}{=} x(x\texttt{<<}e)$$
$$x\texttt{(}e\texttt{~)} \stackrel{\text{def}}{=} x(x\texttt{>~}e) \quad x\texttt{(}e\texttt{>)} \stackrel{\text{def}}{=} x(x\texttt{>>}e)$$

$$\mathit{isticking}(x) \stackrel{\text{def}}{=} x\texttt{<~t == t}$$

$x(\tilde{}e, d) \stackrel{\text{def}}{=}$ `if (`x`<~`e`)==-out then `d` else `$x(\tilde{}e)$

$x(<e, d) \stackrel{\text{def}}{=}$ `if (`x`<<`e`)==-out then `d` else `$x(<e)$

$x(e, d\tilde{}) \stackrel{\text{def}}{=}$ `if (`x`>~`e`)==+out then `d` else `$x(e\tilde{})$

$x(e>, d) \stackrel{\text{def}}{=}$ `if (`x`>>`e`)==+out then `d` else `$x(e>)$

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 somewhat analogous to $x[-1, d]$ in Lola, allowing us to fetch the value of the previous event in stream x, or d if there is not such previous event. The constructors x`(`e`~)`, x`(`e`,`d`~)`, x`(`e`>)`, and x`(`e`>,`d`)` are analogous to their respective past constructors. Finally, $isticking(x)$ indicates if the stream x is producing a value at the current time instant.

Striver offers a concrete syntax where constructors bind stream variables to stream definitions and stream types. Let φ be a Striver specification. We use

```
input type name
```

to indicate that *name* is an input stream of type *type*. This is, $name \in I$ and $T_{name} \stackrel{\text{def}}{=} type \in T$. We use

```
ticks name        := tickexpr
define type name  := valexpr
```

to indicate that *name* is an output stream with tick expression *tickexpr*, type *type*, and value expression *valexpr*. This is, $name \in O$ and $C_{name} \stackrel{\text{def}}{=} tickexpr \in C$, $name \in O$, $V_{name} \stackrel{\text{def}}{=} valexpr \in V$ and $T_{name} \stackrel{\text{def}}{=} type \in T$. We use some syntax highlight to make specifications more readable. Reserved words include `t`, ***`out`*** and ***`notick`*** for which we use italics fonts.

Example 1 The following specification defines a stream y that filters out the negative values of an input stream x. The stream y over-approximates its tick instants as the tick instants of x, and then delegates the filtering to its value expression.

```
input int x
ticks y := x.ticks
define int y := if !isticking(x) then notick
                else if x(~t)<0  then notick
                else x(~t)
```

Example 2 Consider two input event streams: `sale` and `arrival`, where `sale` represents the sales of a certain product, and `arrival` represents the arrivals of the same product to the store. We can define an output event stream `stock` to calculate the stock of that product.

```
input int sale
input 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)
```

Note that `stock` is defined to tick when either `sale` or `arrival` (or both) tick.

Example 3 We can define a stream `clock` to tick periodically from a certain instant onwards using the `delay` operator.

```
ticks clock := {0} U delay 1 clock
define Time clock := 5
```

The stream `clock` produces a value of 5 every 5 time units starting at time 0. Note that this specification has no input streams. □

For a Striver specification $\varphi = \langle I, O, V, C, T\rangle$ to be legal, every ticking expression in C is an α-expression; and every value expression in V is an E-expression. In the next section we show how a simple type inference mechanism guarantees that expressions can be evaluated by the off-the-shelf interpreted theories. If a function application is not applied to a term that guarantees a value of the type needed by the function, the specification is rejected.

3.2 Type inference rules

We use off-the-shelf data domains which do not know about the fresh constants `-out`, `+out` and `notick` introduced to manage the cases of out of stream bounds and absence of tick as values. Therefore, the interpreted function $+$ from the theory *Naturals* is not able to evaluate $(x(\tilde{}t) + y(\tilde{}t))$ in cases where x`(~t)` falls off the trace and becomes $\perp_{\texttt{-out}}$, because *Naturals* does not know about the value $\perp_{\texttt{-out}}$. In Striver we use a simple type system to rule out statically the use of x`(~t)` in $(x(\tilde{}t) + y(\tilde{}t))$ unless it is guaranteed that x`(~t)` is guaranteed to be evaluated to a *Nat* value (typically this is done via an if-then-else in the expression enclosing $(x(\tilde{}t) + y(\tilde{}t))$.

We use $x : \mathcal{E}_D$ to represent that x has been declared of type D. There are three sets of type inference rules, shown in Fig. 1.

- τ inference rules:

  [NOW], [PREVEQ], [PREV], [SUCCEQ] and [SUCC].

  These rules allow inferring that offset expressions generate a time instant or an out-of-stream value.
- The E inference rules:

  [NOTICK], [-OUT], [+OUT], [ACCESS(1)], [ACCESS(2)], [ACCESS(3)], [CONST] and [FUN].

$$\frac{}{\Gamma \vdash \texttt{t} : \mathbb{T}}\ [\textsc{now}] \qquad \frac{}{\Gamma \vdash x\texttt{<<}\tau_z : \mathbb{T}_{\texttt{-out}}}\ [\textsc{Prev}]$$

$$\frac{}{\Gamma \vdash x\texttt{<\~{}}\tau_z : \mathbb{T}_{\texttt{-out}}}\ [\textsc{PrevEq}] \qquad \frac{}{\Gamma \vdash x\texttt{>>}\tau_z : \mathbb{T}_{\texttt{+out}}}\ [\textsc{Succ}]$$

$$\frac{}{\Gamma \vdash x\texttt{>\~{}}\tau_z : \mathbb{T}_{\texttt{+out}}}\ [\textsc{SuccEq}] \qquad \frac{}{\Gamma \vdash \texttt{+out} : \{\bot_{\texttt{+out}}\}}\ [\textsc{+out}]$$

$$\frac{}{\Gamma \vdash \texttt{notick} : \{\bot_{\texttt{notick}}\}}\ [\textsc{notick}] \qquad \frac{}{\Gamma \vdash \texttt{-out} : \{\bot_{\texttt{-out}}\}}\ [\textsc{-out}]$$

$$\frac{\Gamma \vdash x : \mathcal{E}_D \quad \Gamma \vdash \tau_x : \mathbb{T}}{\Gamma \vdash x(\tau_x) : D}\ [\textsc{access}(1)] \qquad \frac{\Gamma \vdash \tau_x : \{\bot_{\texttt{-out}}\}}{\Gamma \vdash x(\tau_x) : \{\bot_{\texttt{-out}}\}}\ [\textsc{access}(2)]$$

$$\frac{\Gamma \vdash \tau_x : \{\bot_{\texttt{+out}}\}}{\Gamma \vdash x(\tau_x) : \{\bot_{\texttt{+out}}\}}\ [\textsc{access}(3)] \qquad \frac{d \text{ is a constant of type } D}{\Gamma \vdash \texttt{d} : D}\ [\textsc{const}]$$

$$\frac{f \text{ is a function with signature } (T_1, \ldots, T_k) \to D \quad \Gamma \vdash E_1 : T_1 \quad \ldots \quad \Gamma \vdash E_k : T_k}{\Gamma \vdash \texttt{f} : (E_1, \ldots, E_k) : D}\ [\textsc{fun}]$$

$$\frac{\Gamma \vdash e_0 = e_1 \quad \Gamma \vdash e_1 : D}{\Gamma \vdash e_0 : D}\ [\textsc{Eq}] \qquad \frac{\Gamma \vdash e : D \oplus \{d\} \quad \Gamma \vdash e \neq d}{\Gamma \vdash e : D}\ [\textsc{NEq}]$$

$$\frac{\Gamma \vdash e = e'}{\Gamma \vdash e' = e}\ [=\textsc{-commut}] \qquad \frac{\Gamma \vdash e \neq e'}{\Gamma \vdash e' \neq e}\ [\neq\textsc{-commut}]$$

$$\frac{\Gamma \vdash e_0 : D_0 \quad \Gamma \vdash e_1 : D_0 \quad \Gamma, \{e_0 = e_1\} \vdash e_2 : D_1 \quad \Gamma, \{e_0 \neq e_1\} \vdash e_3 : D_1}{\Gamma \vdash \texttt{if } e_0\texttt{=}e_1 \texttt{ then } e_2 \texttt{ else } e_3 : D_1}\ [\textsc{Ite}]$$

$$\frac{x \in \Gamma}{\Gamma \vdash x}\ [\textsc{Hip}] \qquad \frac{\Gamma \vdash e : D}{\Gamma \vdash e : D \oplus X}\ [\oplus\textsc{-intro}]$$

$$\frac{\Gamma \vdash e : D \oplus (X \oplus Y)}{\Gamma \vdash e : (D \oplus X) \oplus Y}\ [\oplus\textsc{-assoc}] \qquad \frac{\Gamma \vdash e : X \oplus D}{\Gamma \vdash e : D \oplus X}\ [\oplus\textsc{-commut}]$$

Fig. 1 Type inference rules for Striver.

These rules allow typing value expressions, including stream accesses.

– Type manipulation rules:

[Eq], [NEq], [=-commut], [≠-commut], [Ite], [Hip], [⊕-intro], [⊕-assoc] and [⊕-commut].

These expressions allow accessing the hypotheses as well as introducing and eliminating union types.

A Striver specification $\varphi = \langle I, O, V, C, T \rangle$, in order to be legal, must satisfy that, from the set of type assumptions $\Gamma \stackrel{\text{def}}{=} \bigcup_{x \in I \cup O} \{x : \mathcal{E}_{T_x}\}$, the type inference rules allow deriving that the type of the value expression associated with every output stream is correct: $\forall y \in O, \Gamma \vdash V_y : T_y \oplus \{\bot_{\texttt{notick}}\}$. Also, as mentioned above, every function application $f(e_1, \ldots, e_k)$ of an off-the-shelf data domain must type properly, meaning that all arguments must have the appropriate types required by the f. Otherwise, the specification is declared illegal at compile time.

The type system shown in Fig. 1 is designed to be simple to assess type correctness. More sophisticated type-inference systems would allow inferring the correct typing of expressions that allow writing simpler expressions, but this is outside the scope of this paper.

Example 4 We show now that the stream `stock` from Example 2 has type `int`. The specification without syntactic sugar is:

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

We start from

$$\Gamma \stackrel{\text{def}}{=} \{stock : \mathcal{E}_{int}, sale : \mathcal{E}_{int}, arrival : \mathcal{E}_{int}\}.$$

We first show the type proof of the fact that the following expression is of type `int`.

```
if stock<<t == -out
then 0 else stock(stock<<t)
```

Fig. 2 (a) shows that if `stock<<t` is different from `-out` then `stock(stock<<t)` is of type *int*, with

$$\Gamma^1 \stackrel{\text{def}}{=} \Gamma, \texttt{stock<<t} \neq \texttt{-out}$$

$$\Gamma^2 \stackrel{\text{def}}{=} \Gamma, \texttt{stock<<t} = \texttt{-out}.$$

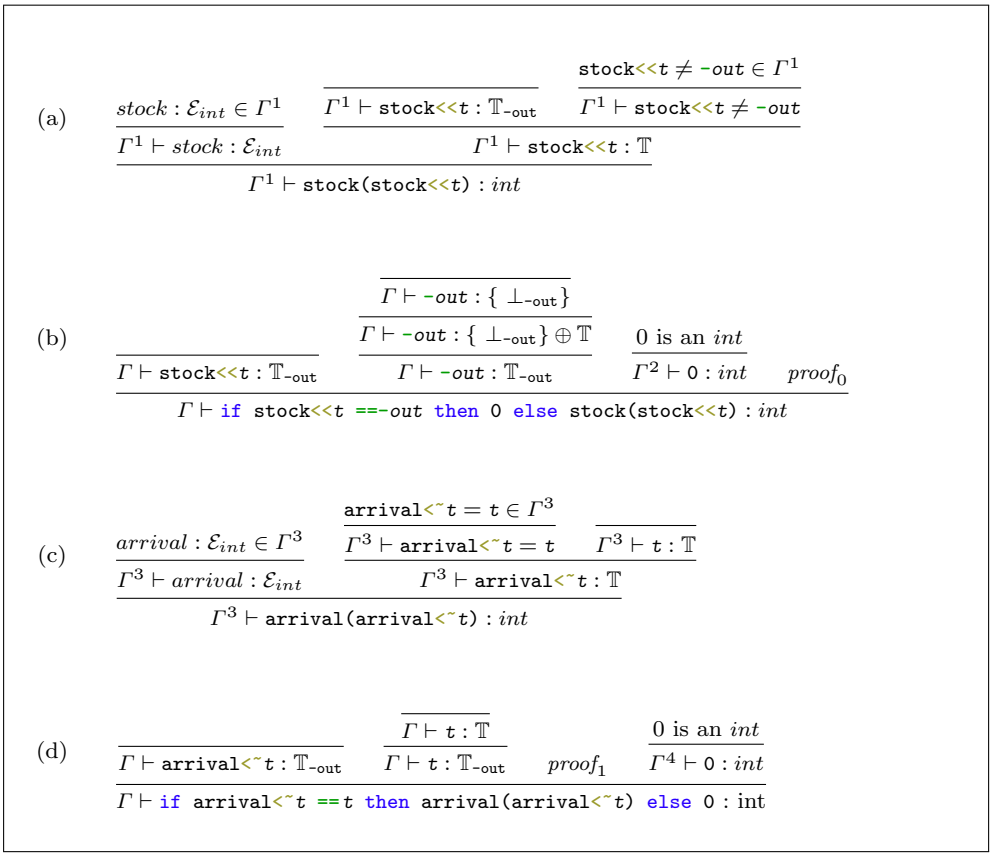

Fig. 2 Type proof trees for Example 4

We call this proof tree $proof_0$.

Then, in Fig. 2 (b), we see the type proof f the whole expression. We then show the proof tree to see that the following expression has type *int*.

```
if arrival<~t == t then arrival(arrival<~t)
```

Fig. 2 (c) shows that if `arrival<~t` is equal to `t`, then `arrival(arrival<~t)` is of type *int*, with

$\Gamma^3 \stackrel{\text{def}}{=} \Gamma, \texttt{arrival<~t} = \texttt{t}$

$\Gamma^4 \stackrel{\text{def}}{=} \Gamma, \texttt{arrival<~t} \neq \texttt{t}.$

We call this proof tree $proof_1$. Next, we see the type proof of the whole expression, in Fig. 2 (d). The proof that the following expression has type *int* is analogous.

```
if sale<~t == t then sale(sale<~t) else 0
```

The types inferred in the previous proofs imply that the applications of $(+)$ and $(-)$, which are of type $(int, int) \to int$, receive the right types. We can conclude that the defining value expression for `stock` has type *int*, and therefore it is also an expression of type $int \cup \{\bot_{\text{notick}}\}$ as required. □

It is easy to show that the type checking described above is decidable (via an easy terminating argument on type inference).

3.3 Semantics

As common in stream runtime verification languages, the semantics of Striver is defined denotationally first. This semantics establishes whether a given input (one stream per input stream variable) and a given output (one stream per output stream variable) satisfy the specification. The semantics of Striver are defined for non-Zeno streams only. We show in Sect. 3.4 that if the input streams are non-Zeno, then the output streams are guaranteed to be non-Zeno as well.

The semantics of Striver can be defined for infinite traces, that is, over time domains that have no $\bar{0}$ or $\bar{1}$ (or neither). However, the absence of each time boundary imposes certain syntactic restrictions over the language. On the other hand, any syntactically well-typed and well-formed specification (see below) can be given semantics if its time domain is finite. As a result, we face with a trade off between language expressivity and time domain restrictions:

1. We can define semantics for unrestricted time domains, but accept only a fragment of the language.

2.a. We can impose the existence of a minimum time $\bar{0}$ and accept a fragment of the language that allows

forwards monitoring (that is, the time domain used in TeSSLa).

2.b. We can impose the existence of a maximum time $\bar{1}$ and accept a different fragment of the language to allow backward monitoring

3. We can impose the existence of both a $\bar{0}$ and a $\bar{1}$ and accept any specification that is syntactically well-typed and well-formed.

In this section, we consider time domains to have a $\bar{0}$ and a $\bar{1}$ and discuss the other cases as extensions. This restriction, combined with the fact that we deal with non-Zeno streams, implies that all streams we consider contain finitely many events. In Sect. 7, we give the syntactic conditions that allow the use of unbounded time domains, effectively enabling us to deal with infinite streams. Note that non-Zenoness is always a necessary condition for our denotational semantics.

This denotational semantics define a satisfaction relation in terms of *valuations*. Given the set of variables $Z = I \cup O$ from the specification, a valuation σ is a map that associates every x of sort D in Z with an event stream from $\mathcal{E}_D$. For a stream variable x, the expression σ_x represents the stream associated with x in σ. Given a valuation σ, we now define the result of evaluating an expression for σ. We define three *evaluation maps* $[\![.]\!]_\sigma$, $[\![.]\!]_\sigma$, $[\![.]\!]_\sigma$ depending on the type of the expression[2]. The evaluation of a ticking expression is a set of ticks. The evaluation of an offset expression is a function that for every point in time, returns another point in time (or $\bot_{\texttt{-out}}$). Finally, the evaluation of a variable expression is a function that for every point in time provides a value of the appropriate domain. These evaluation maps are defined as follows:

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

$$
\begin{aligned}
[\![\{c\}]\!]_\sigma &\stackrel{\text{def}}{=} \{c\} \\
[\![v.\texttt{ticks}]\!]_\sigma &\stackrel{\text{def}}{=} dom(\sigma_v) \\
[\![a_1 \,\mathsf{U}\, \cdots \,\mathsf{U}\, a_k]\!]_\sigma &\stackrel{\text{def}}{=} [\![a_1]\!]_\sigma \cup \cdots \cup [\![a_k]\!]_\sigma \\
[\![\texttt{delay}\ \ \epsilon\ w]\!]_\sigma &\stackrel{\text{def}}{=} \{t' \mid \text{there is a } t \in dom(\sigma_w) \\
&\qquad \text{satisfying } t + \sigma_w(t) = t', \\
&\qquad |\sigma_w(t)| \geq |\epsilon| \text{ and} \\
&\qquad sign(\sigma_w(t)) = sign(\epsilon), \text{ and} \\
&\quad dom(\sigma_w) \cap (t, t') = dom(\sigma_w) \cap (t', t) = \emptyset\}
\end{aligned}
$$

A constant c defines the set of time instants that only contains c. The `ticks` of a stream variable defines the set of ticks that v is assigned in the valuation σ. The union U is interpreted as the union of sets of ticks. Finally, the operator $(\texttt{delay}\ \ \epsilon\ w)$ defines the set of times $t + v$ such that there is an event (t, v) in w, with v of the same sign as ϵ, and $|v| \geq |\epsilon|$; and there is no event between t and $t + v$. Notice that if the stream w produces an event whose value is either of a different sign than ϵ, or is closer to zero than ϵ, then it does not induce a time instant to be added to the set, but it still might prevent the previous value $t + v$ in w to be added to the set.

– *Offset Expressions.* Offset expressions $[\![.]\!]_\sigma$ calculate, given a time instant t, another time instant, or a symbol representing that the limits of the trace were surpassed. The semantics is given in Fig. 3. The interpretation of `t` is the current instant. For $x\texttt{<<}e$, the interpretation is the time of the event in the valuation of x (that is, σ_x) at the closest instant previous to the evaluation of $[\![e]\!]_\sigma$ at the current instant, or the value $\bot_{\texttt{-out}}$ if there is no such event. For $x\texttt{<}\tilde{}\,e$, the interpretation takes the evaluation of $[\![e]\!]_\sigma$ at the current instant, or the previous one at which σ_x contains an event. The semantics of `>>` and `>`$\tilde{}$ are dual to `<<` and `<`$\tilde{}$.

– *Value Expressions.* The semantics of the value expressions are given for an instant t:

$$
[\![x(e)]\!]_\sigma(t) \stackrel{\text{def}}{=} \begin{cases} \bot_{\texttt{-out}} & \text{if } [\![e]\!]_\sigma(t) = \bot_{\texttt{-out}} \\ \bot_{\texttt{+out}} & \text{if } [\![e]\!]_\sigma(t) = \bot_{\texttt{+out}} \\ v & \text{if } [\![e]\!]_\sigma(t) = t' \text{ and } \sigma_x(t') = v \end{cases}
$$

$$
\begin{aligned}
[\![\texttt{f}(E_1, \ldots, E_k)]\!]_\sigma(t) &\stackrel{\text{def}}{=} f([\![E_1]\!]_\sigma(t), \ldots, [\![E_k]\!]_\sigma(t)) \\
[\![\texttt{t}]\!]_\sigma(t) &\stackrel{\text{def}}{=} t \\
[\![\tau_x]\!]_\sigma(t) &\stackrel{\text{def}}{=} [\![\tau_x]\!]_\sigma(t) \\
[\![\texttt{-out}]\!]_\sigma(t) &\stackrel{\text{def}}{=} \bot_{\texttt{-out}} \\
[\![\texttt{+out}]\!]_\sigma(t) &\stackrel{\text{def}}{=} \bot_{\texttt{+out}} \\
[\![\texttt{notick}]\!]_\sigma(t) &\stackrel{\text{def}}{=} \bot_{\texttt{notick}}
\end{aligned}
$$

The interpretation of a stream access for x at e is the value of the stream σ_x at the time t in case the evaluation of e is t and σ_x is defined at t; otherwise it is $\bot_{\texttt{-out}}$ or $\bot_{\texttt{+out}}$. The interpretation of a function is the corresponding interpreted function on the evaluation of the arguments, which includes as particular case the interpretation of constant d (as the value d). We rely on the type system to guarantee that the arguments have values of the right domain. The evaluation of `t` is the current value t. Similarly, the interpretation of offsets is the corresponding instants of time given by the evaluation of offset expression. Finally, the interpretation of `-out` and `notick` is the values $\bot_{\texttt{-out}}$ and $\bot_{\texttt{notick}}$. 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

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

$$[\![\texttt{t}]\!]_\sigma(t) \stackrel{\text{def}}{=} t$$

$$[\![x \texttt{ <<} e]\!]_\sigma(t) \stackrel{\text{def}}{=} \begin{cases} \bot_{\texttt{-out}} & \text{if } [\![e]\!]_\sigma(t) = \bot_{\texttt{-out}} \\ prev_{\leq}(\sigma_x, \bar{1}) & \text{if } [\![e]\!]_\sigma(t) = \bot_{\texttt{+out}} \\ prev_{<}(\sigma_x, [\![e]\!]_\sigma(t)) & \text{otherwise} \end{cases}$$

$$[\![x \texttt{ <}\tilde{}\texttt{ } e]\!]_\sigma(t) \stackrel{\text{def}}{=} \begin{cases} \bot_{\texttt{-out}} & \text{if } [\![e]\!]_\sigma(t) = \bot_{\texttt{-out}} \\ prev_{\leq}(\sigma_x, \bar{1}) & \text{if } [\![e]\!]_\sigma(t) = \bot_{\texttt{+out}} \\ prev_{\leq}(\sigma_x, [\![e]\!]_\sigma(t)) & \text{otherwise} \end{cases}$$

$$[\![x\texttt{>>}e]\!]_\sigma(t) \stackrel{\text{def}}{=} \begin{cases} \bot_{\texttt{+out}} & \text{if } [\![e]\!]_\sigma(t) = \bot_{\texttt{+out}} \\ succ_{\geq}(\sigma_x, \bar{0}) & \text{if } [\![e]\!]_\sigma(t) = \bot_{\texttt{-out}} \\ succ_{>}(\sigma_x, [\![e]\!]_\sigma(t)) & \text{otherwise} \end{cases}$$

$$[\![x\texttt{>}\tilde{}\texttt{ } e]\!]_\sigma(t) \stackrel{\text{def}}{=} \begin{cases} \bot_{\texttt{+out}} & \text{if } [\![e]\!]_\sigma(t) = \bot_{\texttt{+out}} \\ succ_{\geq}(\sigma_x, \bar{0}) & \text{if } [\![e]\!]_\sigma(t) = \bot_{\texttt{-out}} \\ succ_{\geq}(\sigma_x, [\![e]\!]_\sigma(t)) & \text{otherwise} \end{cases}$$

Fig. 3 Semantics of offset expressions.

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 $\sigma_x([\![e]\!]_\sigma(t))$ is defined.

We are now ready to define evaluation models as follows.

Definition 2 (Evaluation model) Given a valuation σ of variables $I \cup O$, the evaluation of the equations for stream $y \in O$ is the event sequence defined as follows:

$$[\![C_y, V_y]\!]_\sigma \stackrel{\text{def}}{=} \{(t,d) \mid t \in [\![C_y]\!]_\sigma \text{ and } d = [\![V_y]\!]_\sigma(t) \text{ and } d \neq \bot_{\texttt{notick}}\}$$

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

In other words, a candidate valuation σ is an evaluation model if σ satisfies all ticking equations and all value equations for all defined stream variables. Note that if an instant t is not in the domain of a stream s, then there is no d to comply with the definition and then $(t,d) \notin \sigma_s$.

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.

Specifications can be ill-defined. For example, the following specification

```
ticks none := {5}
define bool none := not none(~t,False)
```

admits no evaluation model, and the following admits many evaluation models

```
ticks many := {5}
define bool many := many(~t,False)
```

3.4 Dependency graph

Definition 3 states that a specification φ is well defined if for every valuation of the input streams σ_I there is a unique valuation of the output streams σ_O that makes (σ_I, σ_O) an evaluation model. Well-definedness is a semantic condition, which is not easy to check for a given specification (undecidable for expressive enough domains). Following [11,28], we present here a syntactic condition, called well-formedness, that is easy to check on input specifications and guarantees that specifications are well defined.

Given a set of streams Z, we define the subsets of *Present*, *Past* and *Future* offset expressions as the smallest subsets of offset expressions such that:

- $\texttt{t} \in \mathit{Present}$,
- if $e \in \mathit{Future}$ and $x \in Z$, then
  - $(x\texttt{<<}e) \in \mathit{Past} \cap \mathit{Present} \cap \mathit{Future}$,
  - $(x\texttt{<}\tilde{}\texttt{ }e) \in \mathit{Past} \cap \mathit{Present} \cap \mathit{Future}$,
  - $(x\texttt{>>}e) \in \mathit{Future}$, and
  - $(x\texttt{>}\tilde{}\texttt{ }e) \in \mathit{Future}$
- if $e \in \mathit{Present}$ and $x \in Z$, then
  - $(x\texttt{<<}e) \in \mathit{Past}$,

  - $(x\texttt{<\~{}}e) \in \mathit{Present} \cap \mathit{Past}$,
  - $(x\texttt{>>}e) \in \mathit{Future}$, and
  - $(x\texttt{>\~{}}e) \in \mathit{Present} \cap \mathit{Future}$
- if $e \in \mathit{Past}$ and $x \in Z$, then
  - $(x\texttt{<<}e) \in \mathit{Past}$,
  - $(x\texttt{<\~{}}e) \in \mathit{Past}$,
  - $(x\texttt{>>}e) \in \mathit{Past} \cap \mathit{Present} \cap \mathit{Future}$, and
  - $(x\texttt{>\~{}}e) \in \mathit{Past} \cap \mathit{Present} \cap \mathit{Future}$

Note that $e \in \mathit{Future}$ models whether e may be an instant in the future in some valuation. In other words, if $e \notin \mathit{Future}$, then it is guaranteed that $[\![e]\!]_\sigma(t)$ cannot refer to the future of t in any valuation σ.

Definition 4 (Direct dependency) We say that y has a *present* direct dependency on x (and we write $x \xrightarrow{0} y$) if

- $x.\texttt{ticks}$ appears in C_y, or
- V_y contains some present expression $\tau_x \in \mathit{Present}$.

We say that y has a *past* direct dependency on x (and write $x \xrightarrow{-} y$) if

- $\texttt{delay}\ \epsilon\ x$ appears in C_y and $\epsilon > 0$, or
- V_y contains some past expression $\tau_x \in \mathit{Past}$.

We say that y has a future direct dependency on x (and write $x \xrightarrow{+} y$) if

- $\texttt{delay}\ \epsilon\ x$ appears in C_y and $\epsilon < 0$, or
- V_y contains some future expression $\tau_x \in \mathit{Future}$.

In turn, dependencies allow creating a graph with three kinds of edges that represent future, past and present dependencies. This graph is easily computed from the specification and it has linear size in the size of the spec.

Definition 5 (Dependency graph) Given a specification $\varphi = \langle I, O, V, C, T\rangle$ the dependency graph is a directed graph $G_\varphi = (Z, E)$, where set of vertices is $Z = I \cup O$, and set of edges is $E : Z \times Z \times \{\xrightarrow{0}, \xrightarrow{-}, \xrightarrow{+}\}$, where there is an edge $(x, y, t) \in E$ whenever $x \xrightarrow{t} y$ (for $t \in \{=, +, -\}$).

A path in the dependency graph is a *past path* if it contains at least one past dependency edge $\xrightarrow{-}$ and it does not contain any future dependency edge $\xrightarrow{+}$. A path in the dependency graph is a *future path* if it contains at least one future dependency edge $\xrightarrow{+}$ and does not contain any past dependency edge $\xrightarrow{-}$. Note that future paths model paths that necessarily refer to a future time instant, while past paths model paths that necessarily refer to a past instant. If a path is neither future nor past, then it may refer to the current instant. The condition of well-formedness restricts the different kinds of paths in circular dependencies of a given specification.

Definition 6 (Well-formed specifications) A specification φ is well formed if for every maximal strongly connected component (MSCC) M in its dependency graph, either every closed path in M is a *past path* or every closed path in M is a *future path*.

Closed paths are those paths whose initial and final vertices are the same. Closed paths in the dependency graph of a specification φ capture dependencies between a stream and itself. Therefore the fact that all closed paths in a given MSCC are future or past guarantees that no circular dependency can refer to the current instant. In turn, this guarantees that there are no circularities in the information needed to compute the value of a stream at a given instant. The well-formedness condition is easy to check for a given specification and it implies that the dependency graph consists of a DAG of MSCCs, each of which is either future or past.

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

Proof Let φ be a well-formed specification and consider an arbitrary valuation σ_I of the input stream variables of the specification. By assumption, every stream in this input valuation has a finite number of events because they are non-Zeno and the temporal domain is restricted to have a minimum value $\bar{0}$ and a maximal value $\bar{1}$.

We reason by induction in a reverse topological order between the MSCCs, showing that for every MSCC M there is a single valuation of the stream variables in M assuming that all stream variables in lower MSCCs (we call these the inputs to M) have a single valuation. Also, assuming that the input valuations are non-Zeno, the output valuations are also non-Zeno.

Assume that M is a future MSCC (the other case is completely dual). We first define the following "quantum" duration for M as follows:

$$q \stackrel{\text{def}}{=} min\{-\epsilon | (\texttt{delay}\ \epsilon\ w) \in C_x \text{ for any } x, w \in M\}$$

For MSCCs that do not contain a delay, any constant time $q > 0$ can be taken. The definition of quantum for past MSCCs is identical, except that ϵ is used for $-\epsilon$. It is easy to see that the existence of an element in an expression $\texttt{delay}\ \epsilon\ w$ at t only depends on σ_w in the interval $[t + q, \bar{1}]$ because the offset is at least q. We now divide the global duration of the streams $[\bar{0}, \bar{1}]$ into a sequence of $\lceil \frac{\bar{1}}{q} \rceil$ intervals of duration q:

$$[\bar{0}, \bar{0} + q), [\bar{0} + q, \bar{0} + 2q), \ldots, [\bar{0} + nq, \bar{1}]$$

We use I_i for the i-th interval in this sequence. We now reason by induction from $i = n$ down to 0 to show that there are only finitely many candidates $t \in I_i$ that

can be a solution to T_x for some $x \in M$. Take the first atomic expression in T_x for $x \in M$. The case for $s.\texttt{ticks}$ and $\texttt{delay}\ \epsilon\ s$ where $s \notin M$ can only generate a finite number of ticks in I_i because σ_s only has a finite number of ticks by assumption. The case for $\texttt{delay}\ \epsilon\ w$ where $w \in M$ can only generate as many ticks as σ_w has in $\cup_{j>i} I_j$ because the offsets must be at least of q by the definition of q. Finally, the constructs $w.\texttt{ticks}$ and U do not generate new ticks except ticks already included in the previous cases.

Finally, we show that for the finite number of time instants in an interval I_i, the finite number of ticking candidates $\{t_0 < t_1 < \ldots < t_k\}$ in the interval I_i for $[\![T_x, V_x]\!]_{t_j}$ is completely determined for every $t_j \in \{t_0, \ldots, t_k\}$. Note that every closed path in a future MSCC contains at least a future edge. Therefore, removing future edges, the MSCC M becomes a DAG. Evaluating in a reverse topological order $<$ in this DAG guarantees that at time t_j the values of the streams at t_j necessary to compute the value of x at t_j are known. A case inspection in the structure of T_x and V_x reveals that $[\![T_x, V_x]\!]_\sigma$ is completely determined by the events in $\sigma_s|_{[t_j,\bar{1}]}$ if $s < x$, and in $\sigma_s|_{[t_{j+1},\bar{1}]}$ otherwise. We then conclude that there is a unique solution for every σ_x in the interval I_i, which has a finite number of events. Since there is a unique valuation for $\sigma_x|_{[t_j,\bar{1}]}$ for every time instant t_j in every interval I_i, we conclude that there is a unique solution for every σ_x within $[\bar{0}, \bar{1}]$. □

The proof above implies that for every well-formed specification, the input valuation determines uniquely a single valuation σ_x for every stream x. Additionally, in order to determine the value of streams in future MSCCs one only needs to inspect the present and future of streams in the same MSCC, or values of streams in lower MSCCs. Dually, for past MSCCs only the past needs to be inspected. More importantly, the finiteness and acyclicity of the dependencies between events in the evaluation model allow us to reason by induction to prove that operational monitoring algorithms indeed compute the evaluation model.

4 Operational semantics

We show now the operational semantics of Striver. We first present in Sect. 4.1 a monitoring algorithm for the past fragment of Striver, this is, an algorithm to monitor Striver specifications whose dependency graph does not contain positive edges ($\xrightarrow{+}$). This algorithm allows to compute incrementally the output streams from the input streams, and its resources consumption is trace-length independent. Then, in Sect. 4.2 we show a general algorithm for the full version of Striver, which includes also future operators.

4.1 Operational semantics for past specifications

The semantics of Striver specifications introduced in Sect. 3.3 are denotational in the sense that these semantics guarantee that for every input stream valuation there is 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 for the past fragment of Striver. Note that in the past fragment the dependency graph only contains $\xrightarrow{-}$ and $\xrightarrow{0}$ edges. We fix a past specification φ with dependency graph G, and we let $G^{=}$ be its pruned dependency graph (obtained from G by removing $\xrightarrow{-}$ edges). We also fix $<$ to be an arbitrary total order between stream variables that is a reverse topological order of $G^{=}$.

We first present a simple 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 linearly on the size of the specification and not on the length of trace. The modified algorithm is a trace-length independent monitor for past Striver specifications.

The following auxiliary lemma captures sufficient information 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, either $t' \notin dom(\sigma_x)$ and $t' \notin dom(\sigma'_x)$ or $\sigma_x(t') = \sigma'_x(t')$, for every $t' < t$, and*
- *(ii) For every variable x, such that $x \xrightarrow{0}{}^{*} y$, either $t' \notin dom(\sigma_x)$ and $t' \notin dom(\sigma'_x)$ or $\sigma_x(t') = \sigma'_x(t')$, for every $t' \leq t$.*

Then, $\sigma_y(t) = \sigma'_y(t)$.

Proof It is easy to see that $t \in [\![T_y]\!]_\sigma$ if and only if $t \in [\![T_y]\!]_{\sigma'}$, 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'. □

Algorithm 1 PastMonitor: Online Monitor for Past Specifications

```
1:  procedure PastMonitor
2:      H_s ← ⟨⟩ for every s ∈ Z
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 ≠ ⊥_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 ϵ s
18:             switch H_s do
19:                 case ⟨⟩ return ∞
20:                 otherwise
21:                     (t′, v) = latest(H_s)
22:                     if v < ϵ ∨ t′ + v > 1̄ then return ∞
23:                     else return t′ + v
24:         case {c}
25:             if c > t then return c
26:             else return ∞
27:         case a U b
28:             return min(vote(H, a, t), vote(H, b, t))
29:         case y.ticks with y ∈ O
30:             return vote(H, T_y, t)
31:         case s.ticks with s ∈ I
32:             return succ_>(σ_s, t)
```

The online algorithm for the past fragment maintains the following state (H, t_q):

- **History**: H contains a finite event stream 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, which first calculates the next time t_q that is relevant to the monitoring evaluation, and then computes all outputs up to time t_q. We will show that no event can exist 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 $t < t'$ is sufficient.

The core observation that allows the design of our incremental algorithm follows from Lemma 1, which limits the information that is necessary to compute whether stream y at instant t contains some event (t, d) and the value d within the event. 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, PastMonitor, 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 (in the future of the current quiescence time) at which its ticking equation T_y can possibly contain a value. Consequently, no event can possibly be present between the current quiescence time and the lowest vote. Note that recursion at lines 27 and 29 terminates because the graph $G^=$ is acyclic (recall that the specification is well-formed).

If the voted next quiescence time is ∞, it means that all streams have been completed, and thus the algorithm ends. This behavior is reflected in line 6. If the voted next quiescence time is a time instant t, then the algorithm calculates the potential value of each stream at t in topological order $<$ over $G^=$, so the information about the past required in Lemma 1 is contained in H. For every stream, if the calculated potential value is not $\bot_{\texttt{notick}}$, then the event is added to the history of the stream (in line 11) and emitted as an output of the monitor (in line 12).

Note that at every cycle, we need the next event on all input streams at a time instant greater than the current quiescence time. The algorithm will block until all such events occur. As a consequence, the input streams will be inspected and processed at different paces according to the global time. If an input stream s has been consumed completely, then the result of $succ_>(\sigma_s, t)$ will be ∞ at every succeeding cycle. Finally, note that $latest(H_s)$ in line 23 returns the latest event in the past history of stream s (which is guaranteed to be non-empty due to the test in lines 19 and 20).

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 a quiescence time t_q greater than any given t in a finite number of executions of the main loop.

Lemma 2 PastMonitor *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 quiescence 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 that 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 $\lceil \frac{t}{\epsilon} \rceil$ 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 U collects the votes for its sub-expressions, it follows that some sub-expression votes for infinitely many quiescence times in the sequence. The sub-expression cannot be $s.\texttt{ticks}$, 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 ($\texttt{delay}\ \epsilon\ s$). Then, all these votes are caused by different events in H_s that are ticks of s that happened earlier than $t - \epsilon$. □

We finally show that the output of PastMonitor is an evaluation model. We use $H^i_s(\sigma_I)$ 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 $H^*_s(\sigma_I)$ is a finite sequence of events if time is bounded by $\bar{1}$, or if all inputs have a finite number of events and no repetition is introduced in the specification using `delay`. In this case, the vote is eventually ∞ and the monitoring algorithm halts. However, this algorithm can also be used (guaranteeing finite memory) for the continuous online evaluation ad infinitum for unbounded input events or the cyclic generation of events with `delay`.

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 quiescence times is a non-Zeno sequence. We show by induction on the votes of PastMonitor that for every quiescence time t_q, σ is an evaluation model up to t_q, that is $H^*_x|_{t_q} = [\![T_x, V_x]\!]_\sigma|_{t_q}$.

Let t_q^{prev} be a quiescence time and let $t_y = \text{VOTE}(H, y.\texttt{ticks}, t_q^{prev})$. We first show that for every output stream y, $t_y \in [\![T_y]\!]_\sigma$ and for no t' with $t_q^{prev} < t' < t_y$, $t' \in [\![T_y]\!]_\sigma$. This result 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 quiescence 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 quiescence 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. □

Putting together Theorem 2, and Lemmas 1 and 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* PastMonitor*. Then, H^* is the only evaluation model for input σ_I, and H^* is non-Zeno.*

The uniqueness of the evaluation model for a well-formed specification is guaranteed by Theorem 1.

4.1.1 Trace length independent monitoring

The algorithm PastMonitor shown above computes incrementally the only possible evaluation model for a given input stream, but this naive 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 the notion of flat specification. A specification is *flat* if every occurrence of an offset expression is either of the form $x(\texttt{<\~{}t})$ or $x(\texttt{<<t})$. In other words, there can be no nested term of the form $x(\texttt{<\~{}}(y\texttt{<\~{}t}))$ or $x(\texttt{<\~{}}(y\texttt{<<t}))$ or $x(\texttt{<<}(y\texttt{<\~{}t}))$ or $x(\texttt{<<}(y\texttt{<<t}))$. We first show that every specification can be transformed into a flat specification. The flattening applies incrementally the following steps to every nested term $x(E(y\texttt{<<t}))$, where E is an arbitrary offset term:

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

Example 5 Consider a continuous integration process in software engineering, described using the following specification. The intended meaning of stream `faulty` is to report 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,true)
          then notick else commits(<push<<t)
```

After applying the flattening process, the specification becomes:

```
input commit_id commits, unit push, bool tests
ticks faulty             := tests.ticks
define commit_id faulty :=
          if tests(~t,true)
          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 PastMonitor:

- s<~t: the evaluation fetches the value H_s at time t (if s ticks at t) or at the previous ticking time (if s does not tick at t).
- s<<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 (`delay` ϵ s). Hence, to evaluate PastMonitor 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 of φ.*

Proof We apply the flattening step until every output stream definition is flat.

4.2 Operational semantics for full Striver

We now present operational semantics of the full Striver language, including future and past references. As for the algorithm presented in Sect. 4.1, the new algorithm proceeds forward calculating the next event for each output stream. The main idea of this algorithm is to decouple the instant of this calculation for each individual stream. Before, there was a single quiescence time common to all streams, but the new algorithm proceeds with a potentially different time for each stream.

As we did for the past fragment, we show a simple algorithm focusing on simplicity instead of efficiency. However, for the general algorithm shown in this section, there are cases that force the algorithm to maintain an unbounded portion of the calculated history. This is unavoidable as the monitoring problem for future Striver not trace-length independent in general. Some cases can be optimized though. For example, it is easy to see that using the operational semantics presented in this section, a specification whose dependency graph is a tree can be monitored in a trace-length independent manner if the monitor has the additional power of choosing the speed at which each of the input streams is processed. A shared node in the dependency graph means that the events in one stream (say s) influence another (say x) through different sub-expressions e_1 and e_2. It is possible that e_1 and e_2 need events from the common stream s that are arbitrarily far from each other, and between these events there may be an arbitrary number of intermediate events (which cannot be bound a priory) These intermediate events have to be buffered by the engine for their future use. It is work in progress to characterize other classes of specifications that can be monitored in a trace length independent manner, other than those whose dependency graph is a tree.

First, we define the type *Iterator* as $Id \times \mathbb{T}$, whose values are pairs formed by a stream identifier and the timestamp of the last value calculated for the stream. This value corresponds to the time up-to which the stream has been computed which essentially separates queiscience time for every stream. The constants $-\infty$ and $+\infty$ are used to represent out of the bounds (initial and final resp). All stream histories are initialized as empty lists. Given an iterator $it = (s, t)$, we use $it.stream$ for the stream s and $it.time$ for the time t. Given an event $e = (t, d)$, we use $e.time$ for the time t of the event and $e.val$ for the value d.

Monitor. We initialize an iterator for every stream and save it in a map called *outIters*. The monitoring algorithm MONITOR, shown in Algorithm 3, keeps calculating further events for all streams in *outIters*. For each iterator, the algorithm computes the next value, progressing in time. When an event is computed, this is emitted to the environment as a monitor observation. Therefore, for every stream, the events are generated in

Algorithm 3 MONITOR: Online Monitor

```
 1: typedef Iterator :: (Id, 𝕋)
 2: −out ← (−∞, notick)
 3: +out ← (∞, notick)
 4: H_s ← ⟨⟩ for every s
 5: outIters ← ∅
 6: procedure MONITOR
 7:     outIters.add(s, −∞) for every s
 8:     while outIters ≠ ∅ do
 9:         for it ∈ outIters do
10:             (it, ev) ← NEXT(it)
11:             if ev = +out then
12:                 outIters.delete(it)
13:             else
14:                 emit(ev.time, ev.val, it.stream)
15:     end loop
```

increasing time order (note that different streams need not emit the events in increasing order with respect to each other). If the retrieved value for s is +out, this means that σ_s will not contain any more values and the iterator of s is removed from *outIters*.

Algorithm 4 Iterator functions

```
16: function (Iterator, (𝕋, D ∪ {notick})) NEXT(Iterator it)
17:     (s, t) ← it
18:     for i = 0 ... size(H_s) − 1 do
19:         ev ← H_s[i]
20:         if ev.time > t then
21:             return ((s, ev.time), ev)
22:     ev′ ← SOLVENEXT(s)
23:     if last(H_s).val = notick then
24:         removeLast(H_s)
25:     append(H_s, ev′)
26:     return ((s, ev′.time), ev′)
27: function (𝕋, D) ∪ {−out} PEEKPREV(Iterator it)
28:     (s, t) ← it
29:     for i = 0 ... size(H_s) − 1 do
30:         ev ← H_s[i]
31:         if ev.time = t then
32:             if i = 0 then
33:                 return −out
34:             return H_s[i − 1]
```

Algorithm 5 SPEC EXECUTION

```
35: function (𝕋, D ∪ {notick}) SOLVENEXT(s)
36:     if H_s = ⟨⟩ then
37:         t ← −∞
38:     else
39:         t ← last(H_s).time
40:     tv ← CALCULATENEXTTIME(T_s, t)
41:     if tv.val = notick then
42:         return tv
43:     val ← ⟦V_s⟧_H(tv.time)
44:     return ⟨tv.time, val⟩
```

Iterator functions. Iterators are equipped with the following methods, shown in Algorithm 4. The function NEXT returns the next event strictly in the future of the time in the state of the iterator. Such event can be a *progress event*, which means that there are no actual values up to the computed time t, and encodes the processing of the stream up to t in cases where there is no event at t. The simple naive implementation of NEXT shown here loops through the elements in the *History* of the stream, until it finds the first event with a timestamp greater than the iterator's time, and returns the updated iterator, along with the computed event. This event can be already present in the history because the iterators for other streams could have triggered progress in the computation of the history of s. If such an event is not found in the *History*, the function calls SOLVENEXT for the computation of the next event in the valuation of stream s. The event computed is added to the *History* of the stream, removing the last value stored if it was a notick (that is, a progress event), which guarantees that histories can only have a notick as the last event. In other words, progress events are only used to encode the precise quiescence time of the stream. The calculated actual event is returned along with the new state of the iterator. The function PEEKPREV retrieves the previous event before a certain t, or −out if t is the timestamp of the first event in the trace. As a precondition, there has to be an event with timestamp t in the *History* (which is in turn always guaranteed in the algorithm).

Spec execution. The calculation of the next value for a given stream is performed by calling SOLVENEXT, shown in Algorithm 5. If the event history H_s is empty, this computation will call CALCULATENEXTTIME with $-\infty$ as the last timestamp. Otherwise, the computation will use the timestamp of the last event in the history of s. If the event returned by CALCULATENEXTTIME is a progress event (indicated by a notick value), it means that the ticking expression returned progress but no actual event, and this progress event is returned. This also covers the case where the returned event is +out. If a real tick candidate is returned by CALCULATENEXTTIME (indicated by a () value, read as *unit*), this is used to compute the value that corresponds to the denotational semantics of the specification. Note that these semantics may invoke PREV, PREVEQ, SUCC or SUCCEQ according to the expression of V_s, potentially triggering the progress of other stream iterators. An event with the timestamp of the tick and the computed value is returned.

Tick calculation. For the calculation of the next potential tick after t for a `delay` of a stream s with positive

Algorithm 6 TICK CALCULATION

```
45: function (𝕋, {(), notick}) calculateNextTime(expr, t)
46:     switch expr do
47:         case (delay ϵ s)
48:             if ϵ > 0 then
49:                 it ← (s, −∞)
50:                 for (it, ev) ← next(it) do
51:                     if ev.time > t then
52:                         ev′ ← peekPrev(it)
53:                         if ev′ = −out then
54:                             return (ev.time, notick)
55:                         t′ ← ev′.time + ev′.val
56:                         if t′ > 1̄ then
57:                             return +out
58:                         if t′ ≤ ev.time ∧ t′ > t ∧ ev′.val ≥ ϵ
    then
59:                             return (t′, ())
60:                         return (t′, notick)
61:             else
62:                 it ← (s, −∞)
63:                 for (it, ev) ← next(it) do
64:                     if ev = +out then
65:                         return +out
66:                     if ev.time > t ∧ ev.val ≠ notick then
67:                         t′ ← ev.time + ev.val
68:                         if t′ > t ∧ t′ ≥ 0̄ ∧ ev.val ≤ ϵ then
69:                             return (t′, ())
70:                         return (ev.time, notick)
71:         case {c}
72:             if c > t then
73:                 return (c, ())
74:             return +out
75:         case s.ticks
76:             it ← (s, −∞)
77:             for (it, ev) ← next(it) do
78:                 if ev.time > t then
79:                     return (ev.time, ())
80:         case e1 U e2
81:             (t1, v1) ← calculateNextTime(e1, t)
82:             (t2, v2) ← calculateNextTime(e2, t)
83:             if t1 = t2 then
84:                 if v1 = v2 = notick then
85:                     return (t1, v1)
86:                 else
87:                     return (t1, ())
88:             if t1 < t2 then
89:                 return (t1, v1)
90:             return (t2, v2)
```

Algorithm 7 STREAM ACCESS METHODS

```
91: function (𝕋, D) ∪ {−out} prev(s, t)
92:     iter ← (s, −∞)
93:     for (iter, ev) ← next(iter) do
94:         if ev.time ≥ t then
95:             return peekPrev(iter)
96: function (𝕋, D) ∪ {−out} prevEq(s, t)
97:     iter ← (s, −∞)
98:     for (iter, ev) ← next(iter) do
99:         if ev.time = t ∧ ev.val ≠ notick then
100:            return ev
101:        if ev.time ≥ t then
102:            return peekPrev(iter)
103: function (𝕋, D) ∪ {+out} succ(s, t)
104:    iter ← (s, −∞)
105:    for (iter, ev) ← next(iter) do
106:        if ev = +out ∨ (evtime > t ∧ ev.val ≠ notick) then
107:            return ev
108: function (𝕋, D) ∪ {+out} succEq(s, t)
109:    iter ← (s, −∞)
110:    for (iter, ev) ← next(iter) do
111:        if ev = +out ∨ (evtime ≥ t ∧ ev.val ≠ notick) then
112:            return ev
```

delays (a stream with time values $v \geq \epsilon > 0$), the procedure calculateNextTime iterates until it finds an event with a timestamp greater than t. This procedure is shown in Algorithm 6. If the event found is the first event (that is, if peekPrev returns `-out`), then we can conclude that no tick happens up to the time of that event, and this is all we can conclude. If the event found is not the first event, the procedure considers the previous event, and adds its timestamp and its value. If the result is not in the time domain, this is, it is greater than $\bar{1}$, then we output +out. Otherwise, the algorithm checks whether the last event is overridden by the following event. If it is not overridden, the value generated was greater or equal to ϵ, and the corresponding time computed is greater than the argument t, then we output the time of the event along with the unit value. This case corresponds to the `delay` producing an actual tick. If it was overridden, it was lower than ϵ or it has already been output (indicated by the fact that the value is lower or equal to t), then no tick will happen until the time of the next event.

If the expression is a negative `delay` , then we find the next non-progress event in the stream s. If we reach the end of the stream, we output +out. Otherwise, we add the timestamp and value of the event, and check if the result is greater than t, in the time domain (this is greater or equal to $\bar{0}$), and the value was lower or equal to ϵ. If the condition is met, the time of the corresponding event is returned with the unit value. If it is not, then no event will be produced until the timestamp of the next event.

For a `constant` expression, if the constant is greater than t, then it is returned with the unit value. Otherwise, the procedure returns +out.

For a s.`ticks` expression, the procedure iterates over s until it finds the first event with a timestamp greater than t and returns its timestamp along with the unit value.

Finally, for `U`, calculateNextTime finds the lowest timestamp greater than t for each argument stream, and then proceeds in the following way. If the timestamps are the same, and the two values are notick, then a notick at the timestamp is produced. If the times-

tamps are the same, but one of the two values is not notick, then a unit value at the timestamp is produced. Otherwise, the event whose timestamp is the lowest is returned.

Stream access methods. We finally describe the implementation of PREV, PREVEQ, SUCC and SUCCEQ that implement $prev_<$, $prev_\leq$, $succ_>$and $succ_\geq$from the denotational semantics. This is shown in Algorithm 7.

- To calculate $prev_<$, PREV iterates until if finds the first event with a timestamp greater than t on s, and returns the event immediately preceding.
- To calculate $prev_\leq$, PREVEQ iterates until it finds the first event with a timestamp greater or equal to t on s. If such event has timestamp t and it is not a progress event, it is returned. Otherwise, PREVEQ behaves just like PREV.
- To calculate $succ_>$, SUCC iterates until it finds the first non-progress event with a timestamp greater than t on s, or $+\infty$, and returns this event.
- To calculate $succ_\geq$, SUCCEQ iterates until it finds the first non-progress event that has a timestamp greater or equal to t on s, or $+\infty$, and returns it.

The correctness of the algorithm means that the operational semantics implemented by the algorithm outputs all events in the evaluation model for every output stream, and only those events. This is easily shown by induction on the well-formed graph of the finite set of events in the unique evaluation model, guaranteed to exist and be unique by Theorem 1.

5 Comparison with other formalisms

5.1 Comparison with TeSSLa

We compare in this section Striver with the TeSSLa specification language [16]. Even though TeSSLa is defined both for event streams and piece-wise constant signals, event streams and piece-wise constant signals can be easily converted into each other (see, e.g., [22]). We show in this section that TeSSLa can be translated into Striver under the assumptions described below, where the main difficulty is related to the delay operator and the possibility of generating Zeno outputs. Essentially, the decisions in the design of Striver presented in the previous sections guarantee that all outputs are non-Zeno (if all inputs are), while on the other hand, TeSSLa accepts specifications that generate non-Zeno outputs. We modify the delay operator in this section to increase the expressivity of Striver here to be able to cope with these additional specifications.

The design principle of TeSSLa is not to handle explicit time and offsets but instead to offer stream transformers that can be combined to build specifications. A TeSSLa specification consists of a collection of stream variables $Z = I \cup O$ and set of recursive equations of the form $y := e$ with $y \in O$ using the following operators:

$$e ::= \mathbf{nil} \mid \mathbf{unit} \mid x \mid \mathbf{lift}(f)(e, \ldots, e) \mid \mathbf{time}(e) \mid \mathbf{last}(e, e) \mid \mathbf{delay}(e, e)$$

where x is a stream variable. The meaning of **nil** is the empty stream that contains no events. The operator **unit** models the unique stream of type unit that only contains a single event, at time 0. The terminal x allows referring to other streams in the specification. The operators **lift**, **time**, **last** and **delay** are stream transformers, that is, they return streams from other streams. The operator **time** returns a stream that contains the same ticks as the stream passed, except that the values are the instants at which the events occur. The operator **lift** allows using functions from data domains by applying them to the current or previous values of the argument streams. The operator $\mathbf{last}(v, r)$ takes two streams, v for values and r for triggers; **last** returns a stream at the ticking times of r with the previous value of v. Finally, $\mathbf{delay}(d, r)$ takes two streams: a delay stream d and a reset stream r. The output is a stream of type unit that has an event at time t when d has an event $(t - v, v)$, there is no event in r in the time interval $(t - v, t)$, and either there was no tick pending or there is an event in r at instant $t - v$ as well.

We now present a translation from TeSSLa specifications that are non-Zeno to Striver. The set of stream variables is the same, and each equation is translated independently. To simplify the translation we assume that the TeSSLa specification is flat, that is, all arguments of all operators are stream variables. Every specification can be easily flattened by introducing extra variables.

- **nil**: the stream $x := nil$ is translated into:

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

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

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

- **lift**: the stream $x := \mathbf{lift}(f)(s_0, \ldots, s_n)$, where B is the co-domain of f, is translated into:

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

- **time**: the stream $x := \mathbf{time}(s)$ is translated into:

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

– **last**: the stream $x := \textbf{last}(v, r)$, where v is a stream of type A is defined as:

```
ticks x := r.ticks
define A x := v<<t
```

The translation of TeSSLa's `delay` operator is more cumbersome as it allows the possibility of output event streams with diverging (Zeno) time sequences. Since both Striver and TeSSLa assume that inputs are non-Zeno, the only possibility to generate a sequence of Zeno time-stamps is by the `delay` operator generating ticks that are closer and closer. As defined in [16], TeSSLa still allows diverging outputs and classifies as legal those executions that do not diverge. For those specifications and inputs for which the TeSSLa operational semantics diverge the denotational and operational semantics of TeSSLa disagree. A design principle of Striver is to guarantee non-Zeno outputs, which was achieved in Sect. 3 by forcing the time of all delays to be larger than a constant ϵ (which can be arbitrarily small). Hence, since there are non-Zeno sequences in which the `delay` generates arbitrarily close events, the `delay` operator from Sect. 3 is not sufficient to translate TeSSLa to Striver, at least in an inductive way.

To capture all legal TeSSLa specifications, we introduce now a modified delay operator $\texttt{delay}'\ sgn\ f\ w$, where sgn is one of $\{pos, neg\}$, f is a function with type $\mathbb{T} \to Bool$ and w is a stream of type T as before. The intended meaning of f is to tell the delay operator whether to ignore or accept a given event. The semantics are:

$$\begin{aligned}[\![\texttt{delay}\ sgn\ f\ w]\!]_\sigma \stackrel{\text{def}}{=} \{t' \mid\ & \text{there is a } t \in dom(\sigma_w) \\ & \text{satisfying } t + \sigma_w(t) = t', \\ & f(\sigma_w(t)),\ sign(\sigma_w(t)) = sgn \text{ and} \\ & dom(\sigma_w) \cap (t, t') = dom(\sigma_w) \cap (t', t) = \emptyset\}\end{aligned}$$

We require that f be non-divergent for valuation candidate σ, that is, that the set of ticks $[\![\texttt{delay}\ sgn\ f\ w]\!]_\sigma$ is non Zeno. We can mimic the original semantics of Striver's `delay` by choosing $f(v) = |v| \geq |\epsilon|$, which is Zeno-convergent for any valuation. The function f serves as an oracle for delay to accept candidate ticks. The introduction of f imposes an obligation to the writer of the specification, who is now in charge of guaranteeing that f meets the requirement of preventing divergence. If this precondition is not met, then the denotational semantics of the Striver specification is undefined for such an offending trace, and the operational semantics will simply keep producing an ever close set of ticks. If the precondition is met, then the operational and denotational semantics will coincide. Note that this is not really a big practical limitation as a legal f can, for example, let a large number of events be generated and reset the counter if a certain ϵ has been passed since the first event.

With this modified `delay`' operator, we can define the translation of TeSSLa `delay` as follows:

– **delay**: the stream $x := \textbf{delay}(d, r)$ is translated into:

```
ticks x_aux := d.ticks U r.ticks
define Time x_aux :=
  if isticking(d) then
     if isticking(r) ||
        x_aux(<t,0) + (x_aux<<t) <= t
     then d(~t)
     else notick
  else 0

ticks x := delay' 1 (\v -> True) aux
define unit x := ()
```

The choice of $f(v) = true$ as the argument for `delay` allows capturing the semantics of TeSSLa and guarantees that if a given TeSSLa specification is legal (generates non Zeno outputs for every input), then f is convergent for all inputs.

In [22], the authors present a non-blocking engine for a subset of TeSSLa, which can anticipate the computation of some streams even if not all the input streams have events. The operational semantics presented in Sect. 4.1 are kept simple for the sake of explanation, but it can be extended to mimic the asynchronous algorithm in [22] to evaluate and increment iterators independently, and block an iterator only when some of its necessary values is not present.

Theorem 4 *The semantics of a legal TeSSLa specification φ, and the Striver specification φ' resulting from following the shown translation over φ are equal over any valid input.*

Proof (sketch) An easy induction on the structure of TeSSLa expressions allows proving that the resulting Striver specification obtained is equivalent. We sketch a proof for the case of **delay**. The output stream x generates the unit value after the last alarm set by x_aux, with a new value on x_aux overriding the pending alarm. The auxiliary stream x_aux behaves as follows: If stream d is generating an event, then x_aux decides whether it will emit the value or ignore it. If the reset stream is also generating a value, the delay is emitted. Alternatively, if the last alarm set by x_aux has already gone off (or if no alarm was ever set), then the delay is emitted. If there was a pending alarm and the reset stream is not generating a value, the delay is ignored, and x_aux produces a `notick` value. If instead

the delay stream d is not producing a value (and thus, the reset stream r is), then we cancel any pending alarm by emitting a 0. □

5.2 Comparison with signal temporal logic

We now show how Striver also subsumes the signal temporal logic (STL) [23,24]. To do so, we extend the core Striver language with new constructors to allow the definition of properties over sliding windows without the need to introduce container-types in a given data theory. First, we introduce the notion of *carried values* which essentially allow tagging the timestamp at which a stream ticks with values from the ticking subexpression that actually causes the tick. By itself, this extension does not add expressive power to the language, and could be expressed using the instruments presented in Sect. 3. The second extension introduces *shift*, which allows moving the ticks of another stream by a positive or negative constant. Together with the carried values, shift allows defining a shift stream transformer and truly sliding windows. In detail, we extend the Striver language with two new capabilities:

1. **Carried values**: Every time a value expression is computed at an instant t, it is because t belongs to its associated ticking expression, which in turn can be caused by other streams, by a constant or by a delay. The idea of the *carried value* is to give a way to access the values of the ticking streams from the value expressions. To achieve this, we enrich the evaluation context of the value expressions with a new language construct `cv` whose value is a tuple containing the values of the members in the ticking expression that induced the tick. We indicate the type of the *carried value* in the tick expression of the stream. In particular, if the n-th source of ticks is not producing a value at the time of evaluation, then the n-th element of `cv` is $\bot_{\texttt{notick}}$.
2. **Shift**: The constructor `shift` extends ticking expressions allowing to shift a stream by a constant duration. Together with carried values, we can trivially shift a stream by a given `length` as follows:

```
input  int s
ticks  int shift_s := shift 3sec s
define int shift_s := cv
```

Note that the carried value `cv` allows fetching the value of `s` at a different time (after the delay `3sec`), which otherwise would require calculating the time instant and allow accessing values of streams at arbitrary times. Such a feature would increase the complexity of monitoring algorithms and typically requires preserving the whole trace of every stream.

We show how the addition of these constructors affects the syntax:

- *Ticking Expressions*: We add the `shift` operator to the ticking expressions.

$$\begin{aligned} \alpha &::= \alpha' \mid \alpha' \,\mathsf{U}\, \cdots \,\mathsf{U}\, \alpha' \\ \alpha' &::= \{c\} \mid v.\texttt{ticks} \mid \texttt{delay}\ \epsilon\ w \mid \texttt{shift}\ c\ v \end{aligned} \tag{1}$$

- *Offset Expressions*: Offset expressions are not affected by these extensions.
- *Value Expressions*: We add the constructor to access the *carried values*.

$$E := d \mid x(\tau_x) \mid \texttt{f}(E_1, \ldots, E_k) \mid \tau' \mid \texttt{-out} \mid \texttt{+out} \mid \texttt{notick} \mid \texttt{cv} \tag{2}$$

The additional expression `cv` represents the value carried by the ticking expression.

The idea is that the ticking expressions calculate not only a set of tick instants, but also a value associated with every potential tick instant. For the ticking expression $v.\texttt{ticks}$, the associated value at instant t is $\sigma_v(t)$. For the ticking expression c, the associated value for each instant t is () (the unit value). For the ticking expression $\texttt{delay}\ \epsilon\ w$, the associated value for instant t is the value of w that made t become a tick instant. For the ticking expression $\texttt{shift}\ c\ v$, the associated value for each instant t is $\sigma_v(t-c)$. Finally, for the ticking expression $e_0\ \mathsf{U}\ \cdots\ \mathsf{U}\ e_k$, the associated value for each instant t is a k-tuple where each element $i = 1, \cdots, k$ is the value carried by the expression e_i if t is in the tick expression e_i, and $\bot_{\texttt{notick}}$ otherwise.

This value carried by the tick expression can be accessed from the value expression with the new constructor `cv`.

Note that if the tick expression does not have a `shift` expression, we can calculate the carried value using the original value operators using the construct `isticking(r)` and `r(~t)` over every non-delayed tick expression of the form $r.\texttt{ticks}$, and calculating with `r(<t)` over every $\texttt{delay}\ \epsilon\ r$ tick expression.

We are now ready to show that Striver with these extensions subsumes signal temporal logic (STL) [23, 24]—when interpreted over piecewise-constant signals. Piecewise-constant signals are signals that only change value in finitely many points in every given interval, and remain constant between two points of value change. The syntax of STL is

$$\varphi ::= \mathit{true} \mid \mu_f \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi\, \mathcal{U}_{[a,b]}\, \varphi$$

where f is a function from $\mathbb{R}^n$ to $\mathbb{R}$, and a and b belong to the temporal domain. The satisfaction relation

is defined over a sequence x of real valued signals and a time-point t as follows.

$$
\begin{array}{lll}
(x,t) \models \mathit{true} & & \text{always holds} \\
(x,t) \models \mu_f & \text{iff} & f(x[t]) > 0 \\
(x,t) \models \neg\varphi & \text{iff} & (x,t) \not\models \varphi \\
(x,t) \models \varphi_1 \vee \varphi_2 & \text{iff} & (x,t) \models \varphi_1 \text{ or } (x,t) \models \varphi_2 \\
(x,t) \models \varphi_1 \,\mathcal{U}_{[a,b]}\, \varphi_2 & \text{iff} & \text{for some } t' \in [t+a, t+b], \\
 & & \quad (x,t') \models \varphi_2, \\
 & & \text{and for all } t'' \in [t,t'], \\
 & & \quad (x,t'') \models \varphi_1
\end{array}
$$

Event streams have a alternative interpretation as piecewise-constant signals, where the signal changes at the points at which events are produced. A translation of STL into Striver like the one shown in this section enables the encoding of quantitative semantics of STL by enriching the data types of expressions and verdicts. We show here how to check STL Boolean properties by translating them to Striver specifications. The specification is constructed recursively over φ:

- **True:** We translate *true* as the stream `tr` as follows:

```
output bool tr
ticks  unit tr := {0}
define bool tr := true
```

- **Function sampling μ_f:**. We assume that the input signal is `input D x`, and define the Boolean output signal `mu_f` as follows:

```
output bool mu_f
ticks  D    mu_f := x.ticks
define bool mu_f := f(cv) > 0
```

- **Negation:** Given the Boolean stream `x` for φ, we define the stream `neg_x` for $\neg\varphi$ as follows:

```
output bool neg_x
ticks  bool neg_x := x.ticks
define bool neg_x := not cv
```

- **Disjunction:** Given the Boolean streams `x` and `y`, for φ_1 and φ_2 (resp), we define the stream `x_or_y` for $(\varphi_1 \vee \varphi_2)$ as follows:

```
output bool          x_or_y
ticks (bool,bool) x_or_y:=x.ticks U y.ticks
define bool          x_or_y:=
         x(~t,false) || y(~t,false)
```

- **Until:** Given Boolean streams `x` and `y` for φ_1 and φ_2 (resp), and given `a` and `b` we define the stream `x_U_y` for $(\varphi_1 \,\mathcal{U}_{[a,b]}\, \varphi_2)$ as follows:

```
output bool x_U_y

ticks  bool shift_y_a := shift -a y
define bool shift_y_a := cv

ticks  bool shift_yT_a := shift_y_a.ticks
define Time shift_yT_a :=
  if cv then t else notick

ticks  bool x_F:=x.ticks
define Time x_F:=if cv then notick else t

ticks (bool,bool,bool,bool) x_U_y :=
  shift -a y U shift -b y
  U shift -b x U x.ticks
define bool x_U_y := let
  min_y := if shift_y_a(~t,false) then t
                 else shift_yT_a(>t,infty)
  min_xF := if !x(~t,false)
            then t else x_F(>t,infty) in
  min_y+a <= t+b && min_y+a < min_xF
```

  Essentially, the intermediate stream `shift_y_a` defines a shift of `y` by exactly a time units, and then `shift_yT_a` filters out the events with value *false* and keeps occurrences of value *true* only. Hence, the time and value of the next event in `shift_yT_a` at any instant t correspond to the next time `y` becomes true after $t+a$. The expression `min_y` contains the earliest time at which `y` becomes true (considering the possibility of `t` itself if `y` is already true). Similarly, `min_xF` contains the earliest time at which `x` becomes false (considering the possibility of `t` itself if `x` is already false). With these auxiliary definitions, the value expression of `x_U_y` simply checks that `y` becomes true within $[a,b]$ and that `x` is true from `t` up-to that point.
  The tick expression of the stream indicates the times at which its value can change, namely when a `y` event enters or leaves the sliding window defined by $[t+a, t+b]$, or when a `x` event enters or leaves the sliding window defined by $[t, t+b]$.

We use the constant `infty` to represent a value that is greater than any value of $\mathbb{T}$. The translation presented is bottom-up and simple, but it does not exploit the fact that, if b is not ∞, only a bounded future must be explored. As it is, the translation of until is a future specification. We extend now the language with *bounded* offset operators by redefining τ-expressions as follows:

$$
\begin{array}{l}
\tau_x ::= x\texttt{<}\tilde{}\,\tau' \mid x\texttt{<<}\tau' \mid x\texttt{>}\tilde{}\,\tau' \mid x\texttt{>>}\tau' \mid x\texttt{<}\tilde{}_b\tau' \mid \\
\qquad x\texttt{<<}_b\tau' \mid x\texttt{>}\tilde{}_b\tau' \mid x\texttt{>>}_b\tau' \\
\tau' ::= \texttt{t} \mid \tau_z \qquad \text{for } z \in Z
\end{array}
$$

The semantics of the newly added operators is as follows: Considering $\forall x \in \mathbb{T}. \bot_{\texttt{-out}} < x$ and $x < \bot_{\texttt{+out}}$,
If $[\![x\texttt{<}\tilde{}\,e]\!]_\sigma(t) < [\![e]\!]_\sigma(t) - b$, then $[\![x\texttt{<}\tilde{}_b e]\!]_\sigma(t) = \bot_{\texttt{-out}}$.
Otherwise, $x\texttt{<}\tilde{}_b e$ behaves as $x\texttt{<}\tilde{}\,e$ at t.
If $[\![x\texttt{<<}e]\!]_\sigma(t) < [\![e]\!]_\sigma(t) - b$, then $[\![x\texttt{<<}_b e]\!]_\sigma(t) = \bot_{\texttt{-out}}$.
Otherwise, $x\texttt{<<}_b e$ behaves as $x\texttt{<<}e$ at t.
If $[\![x\texttt{>}\tilde{}\,e]\!]_\sigma(t) > [\![e]\!]_\sigma(t) + b$, then $[\![x\texttt{>}\tilde{}_b e]\!]_\sigma(t) = \bot_{\texttt{+out}}$.
Otherwise, $x\texttt{>}\tilde{}_b e$ behaves as $x\texttt{>}\tilde{}\,e$ at t.
If $[\![x\texttt{>>}e]\!]_\sigma(t) > [\![e]\!]_\sigma(t) + b$, then $[\![x\texttt{>>}_b e]\!]_\sigma(t) = \bot_{\texttt{+out}}$.
Otherwise, $x\texttt{>>}_b e$ behaves as $x\texttt{>>}e$ at t.

Mathematically, the semantics of $x\texttt{<~}_b e$ is the following:

$$[\![x\texttt{<~}_b e]\!]_\sigma(t) \stackrel{\text{def}}{=} \begin{cases} \bot_{\texttt{-out}} & \text{if } [\![x\texttt{<~}e]\!]_\sigma(t) = \bot_{\texttt{-out}} \\ & \text{or } [\![x\texttt{<~}e]\!]_\sigma(t) < [\![e]\!]_\sigma(t) - b \\ [\![x\texttt{<~}e]\!]_\sigma(t) & \text{otherwise} \end{cases}$$

The semantics of the other operators are analogous. Even though these expressions do not enhance the expressive power of Striver, they enable the monitoring engine to stop seeking a value if the time progress grows beyond a bound. With this information, the engine can optimize the execution and even guarantee trace-length independence assuming that the event rate is bounded. An empirical study on how using bounded operators affects the resource performance is included in Sect. 6. We can now use the $\texttt{>>}_b$ operator to define a more efficient version of STL's *Until*:

```
ticks (bool,bool,bool,bool) x_U_y :=
    shift -a y U shift -b y
    U shift -b x U x.ticks
define bool x_U_y := let
  min_y  := if shift_y_a(~t,false) then t
            else shift_yT_a(>t_{b-a},infty)
  min_xF := if !x(~t,false) then t
            else x_F(>_b t,infty) in
  min_y+a <= t+b && min_y+a < min_xF
```

At instant t, the implementation will be keeping the events in `y` in the range $[t+a, t+b]$ and the events in `x` in the range $[t, t+b]$.

6 Empirical evaluation

In this section, we report on an empirical evaluation of Striver. We conduct two sets of experiments: one for the past fragment of Striver, that is guaranteed to run in bounded resources, and the other for the full version of Striver, including the extension shown in Section 5.2. All experiments were executed on a virtual machine running on an Intel Xeon at 3GHz with 32GB of RAM.

6.1 Past Striver

The empirical evaluation of past Striver is based on an implementation written in the Go programming language[3] which is the core element of the Elastest Monitoring Service[4]. We run experiments to measure the memory usage and time per event for two collections of specifications:

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

[4] Available at `https://github.com/elastest/elastest-monitoring-service`

– The first collection generalizes Example 2 computing the stocks of p independent products. These specifications contain a number of streams proportional to p, where each defining equation is of constant size. Even though each output stream in the specification could be monitored in parallel, our engine is completely sequential.

```
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)
```

– 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. The resulting specifications have depth proportional to k.

```
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
```

We instantiate k and p from 10 to 500 and run each resulting specification with a set of generated input traces. We 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 (shown in the top two plots in Fig. 4). 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 throughput of events processed is independent of the length of the trace, and is a constant in

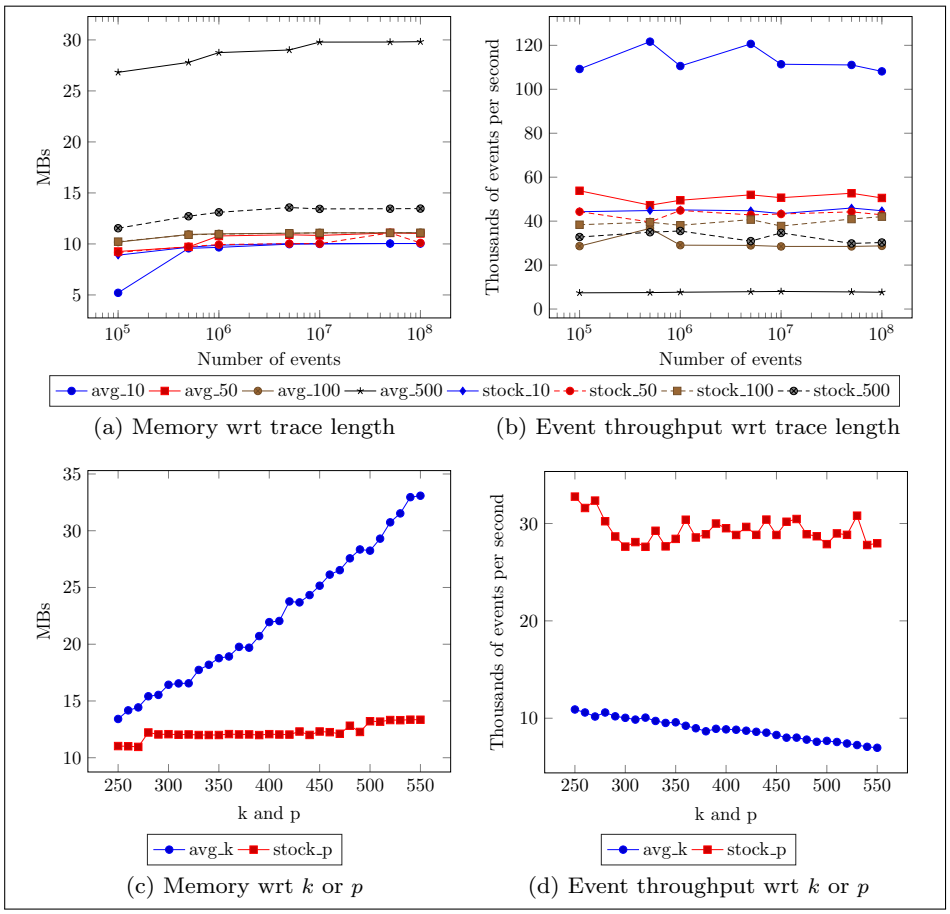

Fig. 4 Empirical evaluation of the past fragment of Striver

the thousands of events per second in each experiment. 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.

6.2 Full Striver

We report an empirical evaluation of a prototype sequential Striver implementation, written in the Java programming language[5]. For this set of experiments we consider a simple STL specification of a moving vehicle. The speed of the vehicle is an input stream of type `double`. The property to specify is: "*Whenever the vehicle is moving* too fast, *it must decelerate continuously until it reaches a* safe speed *within 5 s.*" We say that the vehicle is moving *too fast* if its speed is greater than 1, and we define a *safe speed* as a speed under 0.8. We can write this property in STL as follows:

$$\varphi : (speed > 1) \rightarrow \left(decel \, \mathcal{U}_{[0,5]} \; speed < 0.8\right)$$

Note that this specification requires a Boolean input signal *decel* that indicates whether the vehicle is decelerating. We translate this property into Striver using the following specification, where `slow_down` is obtained by translating the Until operator as shown in Sect. 5.2.

```
input  double speed

ticks  double toofast := speed.ticks
define bool   toofast := cv > MAX_SPEED

ticks  double speedok := speed.ticks
define bool   speedok := OK_SPEED > cv

ticks  double decel := speed.ticks
define double decel := cv > speed(>t)

slow_down := decel U_[0,5] speedok
```

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

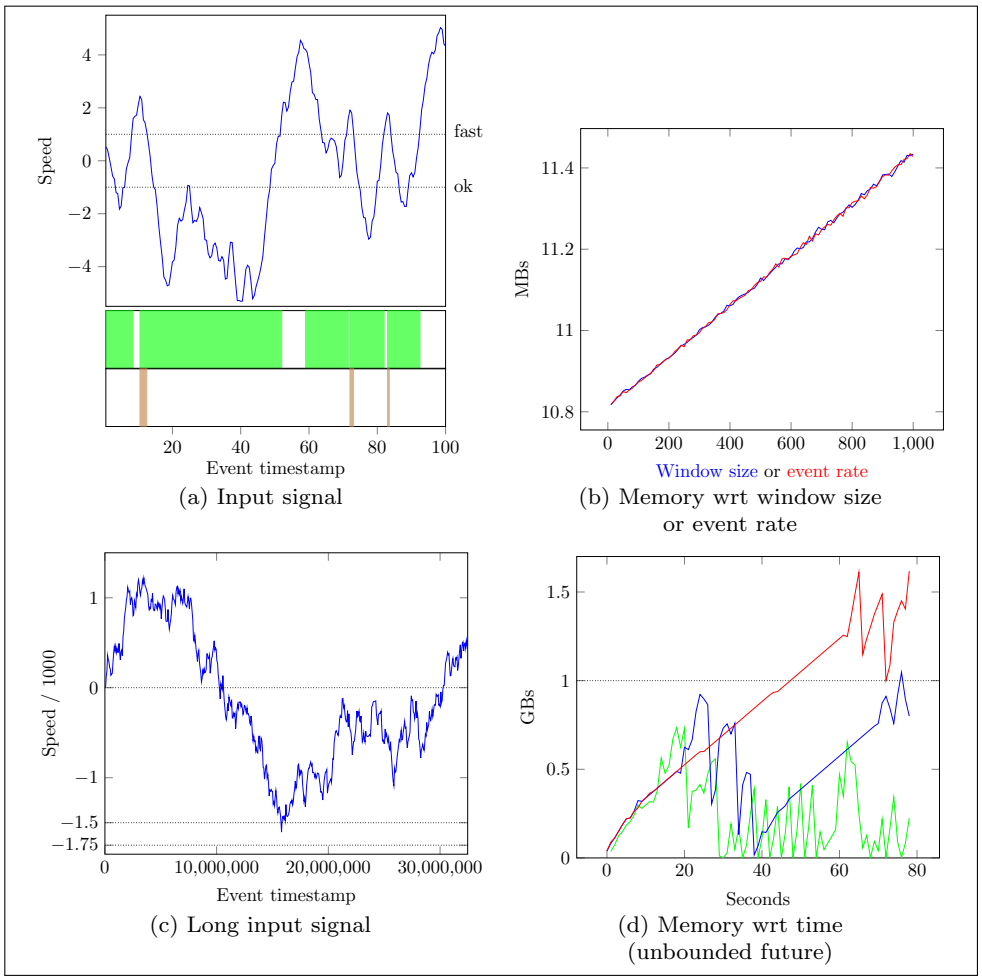

Fig. 5 Empirical evaluation of the fully fledged version of Striver

```
ticks  (bool,bool) ok :=
  toofast.ticks U slow_down.ticks
define bool        ok :=
  toofast(~t,false) => slow_down(~t,true)
```

Using Striver we measure the deceleration with the signal speed comparing its value at the current instant with the next value, and thus there is no need of an extra input signal.

The input data, shown in Fig. 5a, c, are generated pseudo-randomly. Figure 5a shows an input illustrating under the graph (in green) the regions where φ holds. Also, at the bottom we show the regions where the speed is too high, but where the car decelerates continuously until reaching a safe speed, within 5 time units. Fig. 5c shows a much longer input signal.

We translated the specification using bounded future operators, and showed that the memory consumption of the monitor remains constant over the trace length. However, the memory requirement correlates linearly with the size of the window and with the input event rate, as can be seen in Fig. 5b.

On the other hand, if we use a Striver specification with unbounded future operators, then the memory requirement depends on the input signal and is no longer solely determined by the window size and input event rate. The memory usage of the different runs can be seen in Fig. 5d. In the Striver specification with unbounded future accesses, the *Until* expression needs to retrieve at a given time, the next instant at which the vehicle *decelerates* and also the next instant at which its speed is *safe*. If one of these instants is far in the future, the monitor engine needs to consume and store all the input up to that point. This causes a rapid increase in the memory consumption and is completely dependent on the shape of the input signal, which is in principle arbitrary. Once the relevant instants are found, the input signal is consumed up to that point, which causes a rapid decrease in memory consumption. As a result of this, we observe that memory usage presents peaks during the execution. The green curve in Fig. 5d represents the memory consumption of a run with the original *safe speed*, of 0.8. We can see that usually this

value is reached quickly and keeps the memory below the maximum value, which was set at $1GB$.

In the blue run, we modified the *safe speed* to -1600. This value requires more time to find the relevant time instants, resulting in higher peaks of memory usage, that go down at approximately time instant 40. From that point on, the speed of the car never comes back down to a value as low as -1600, and thus the memory consumption reaches the maximum memory.

The red curve represents the memory consumption of a run with a *safe speed* of -1750. Since this value is never reached, or it is reached too far away in the input trace, the memory consumption goes up continuously well beyond the maximum memory threshold (without producing an output), and eventually the program crashes with a `java.lang.OutOfMemoryError`. Note how a small variation in the input data yields very different memory consumption curves for unbounded future specifications.

7 Final discussions and conclusion

We have introduced Striver, a stream runtime verification specification language for timed event streams, equipped with explicit time. We have presented a trace-length independent online monitoring algorithm for the past fragment, and we show empirically that it behaves as expected in terms of the bounds of resources. We have also presented an online monitoring algorithm for the full version of the language and we show how its future fragment needs not be trace length independent. We have shown how to translate specifications in other specification languages, such as TeSSLa and STL, to Striver and we showed empirically that the memory requirement to monitor an STL specification is bounded by the ratio between input rate and the size of the intervals in the usage of *Until* operators.

Unbounded time domains. The semantics of Striver introduced in Section 3.3 requires temporal domains to have both a minimal element $\bar{0}$ and a maximal element $\bar{1}$ for all specifications to be well-defined in the general case. We describe here how to relax these requirements. The boundaries in the time domain are necessary for two reasons:

- First,is to provide a base case for recursion, which is required by closed paths in the dependency graph. A negative cycle imposes the need for a $\bar{0}$, while a positive cycle imposes the need for a $\bar{1}$. For example, consider the following specification:

```
input unit r
ticks s := r.ticks U {5}
define int s := s(>t,0) + 1
```

  There must be a value for s at 5, but if r produces infinitely many events, it is not possible to find an integer that satisfies the value equation of s.
- Second, to prevent faulty references to the occurrences of functions $prev_<$, $prev_\leq$, $succ_>$, and $succ_\geq$, in the semantics of τ expressions. An expression s<<e or s<~e that uses an expression e whose value could be $\bot_{\texttt{+out}}$ (for example r>>t) imposes the need for a $\bar{1}$, while a s>>e or s>~e using an expression e whose value could be $\bot_{\texttt{-out}}$ imposes the need for a $\bar{0}$. For example, consider the following specification:

```
input unit r
ticks v := {0}
define void v := notick
ticks s := {0}
define Time s := r<<(v>>t)
```

  The stream s at 0 is trying to fetch the last value of r, which may not exist if r produces infinitely many events.

If we are dealing with a specification whose dependency graph does not have positive cycles, and there are no expressions s<<e or s<~e where the semantics of e could be $\bot_{\texttt{+out}}$, then there is no need to impose the presence of a $\bar{1}$ in the time domain. For example, this is the case when one uses only past expressions. Dually, if we are dealing with a specification whose dependency graph does not have negative cycles, and there are no expressions s>>e or s>~e where the semantics of e could be $\bot_{\texttt{-out}}$, then there is no need to impose the presence of a $\bar{0}$ in the time domain.

The operational semantics of the past-only fragment of Striver presented in Sect. 4.1 always converges to the correct evaluation model if the specification is well-defined. On the other hand, the operational semantics of the full language presented in Sect. 4.2 may diverge for well-defined specifications which do not contain positive cycles, do not define a $\bar{1}$, but use unbounded future accesses or negative delays. For example, the following specification is well defined:

```
ticks (unit, unit) clock:={0} U shift 10 clock
define unit        clock:=()

ticks  unit empty  := clock.ticks
define void empty  := notick

ticks  unit empty2 := {0}
define void empty2 := empty(>t,notick)
```

The stream `clock` generates a `()` every 10 time units starting at 0, and the streams `empty` and `empty2` have no events, something that can be easily deduced from their types. A smart compiler could realize that `empty` and `empty2` have no events from their types or checking that the value expression of `empty` is `notick`, but an immediate application of the operational semantics

algorithm will diverge trying to calculate the value of the stream `empty2` at time 0. The main reason is that a future access (like `empty(>t,notick)`) requires to compute the next ticking instant of `empty`. Even though this is a single offset, this instant may be arbitrarily far in the future, or not even exist. Therefore, in a model of time where the future is unbounded, this computation may not terminate.

Language duality. There exists a duality between the past and the future operators of Striver, which can be exploited to offline monitor any specification using the online algorithm presented in Sect. 4.1 and following the idea of multiple passes used in Lola [11,28]. We start from a well formed specification and partition its dependency graph into maximal strongly connected components (MSCC). Since the specification is well formed, each of the MSCCs either has only positive cycles, or have only negative cycles. MSCCs with negative cycles will be computed from the beginning of the trace forwards. MSCCs with positive cycles will be computed from the end of the trace backwards. Every pass will store the output streams of the MSCCs being calculated in a local file storage. The dependency graph of MSCCs is actually a DAG, and the order of evaluation is a reverse topological order using the order induced by the dependencies. Independent MSCCs can be computed in parallel. If the specification does not contain both positive and negative cycles, then the whole specification can be computed using only one pass. The lack of cycles of any kind implies that well-definedness is not dependent on boundaries over the time domain, and that the direction of the computation can be freely chosen by the algorithm. In particular, the STL translation of a property yields a dependency graph with no cycles, and thus can be computed both forward or backwards as desired.

Future work. Future work includes the extension of the language with parametrization, (like in quantified event automata (QEA) [29], MFOTL [30] and Lola2.0 [31]), to dynamically instantiate monitors for observed data items. We also plan to study offline evaluation algorithms, and algorithms that tolerate deviations in the time-stamps and asynchronous arrival of events from the different input streams, computing the values of the output streams as soon as the necessary data are available.

References

1. Klaus Havelund and Allen Goldberg. Verify your runs. In *Proceedings of VSTTE'05*, LNCS 4171, pages 374–383. Springer, Berlin, 2005.
2. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.
3. *Lectures on Runtime Verification—Introductory and Advanced Topics*, volume 10457 of *LNCS*. Springer, 2018.
4. Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In *Proceedings of TACAS'02*, LNCS 2280, pages 342–356. Springer, Berlin, 2002.
5. Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In *Proceedings of CAV'03*, volume 2725 of *LNCS 2725*, pages 27–39. Springer, Berlin, 2003.
6. Andreas Bauer, Martin Leucker, and Chrisitan Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Soft. Eng. Methods*, 20(4):14, 2011.
7. Koushik Sen and Grigore Roşu. Generating optimal monitors for extended regular expressions. *ENTCS*, 89(2):226–245, 2003.
8. Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *J. ACM*, 49(2):172–206, 2002.
9. Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *Proceedings of VMCAI'04*, LNCS 2937, pages 44–57. Springer, Berlin, 2004.
10. Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Autom. Softw. Eng.*, 12(2):151–197, 2005.
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 *Proceedings of TIME'05*, pages 166–174. IEEE, New York, 2005.
12. Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A hard real-time runtime monitor. In *Proceedings of RV'10*, LNCS 6418. Springer, Berlin, 2010.
13. Alwyn E. Goodloe and Lee Pike. Monitoring distributed real-time systems: A survey and future directions. Technical report, NASA Langley Research Center, 2010.
14. Laura Bozelli and César Sánchez. Foundations of Boolean stream runtime verification. In *Proceedings of RV'14*, volume 8734 of *LNCS*, pages 64–79. Springer, Berlin, 2014.
15. Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. In *Proceedings of FM'06*, LNCS 4085, pages 573–586. Springer, Berlin, 2006.
16. Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. TeSSLa: temporal stream-based specification language. In *Proceedings of the 21st. Brazilian Symp. on Formal Methods (SBMF'18)*, LNCS. Springer, Berlin, 2018.
17. Peter Faymonville, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. Real-time stream-based monitoring. *CoRR*, arXiv:1711.03829, 2017.
18. Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):642–657, 1999.
19. Francisco Gortázar, Micael Gallego, Boni García, Giuseppe Antonio Carella, Michael Pauls, and Ilie-Daniel Gheorghe-Pop:. ElasTest–an open source project for testing distributed applications with failure injection. In *Pro-*

ceedings of the 2017 IEEE Conf. on Network Function Virtualization and Software Defined Networks (NFV-SDN'17), pages 1–2. IEEE, New York, 2017.
20. Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symp. on Foundations of Computer Science (FOCS'77)*, pages 46–67. IEEE Computer Society Press, New York, 1977.
21. Felipe Gorostiaga and César Sánchez. Striver: stream runtime verification for real-time event-streams. In *Proceedings of the 18th Int.'l Conf. on Runtime Verification (RV'18)*, volume 11237 of *LNCS*, pages 282–298. Springer, Berlin, 2018.
22. Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm. TeSSLa: runtime verification of non-synchronized real-time streams. In *Proceedings of the 33rd Symposium on Applied Computing (SAC'18)*. ACM, New York, 2018.
23. Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *Proceedings of FORMATS/FTRTFT 2004*, volume 3253 of *LNCS*, pages 152–166. Springer, Berlin, 2004.
24. *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, Berlin, 2018.
25. Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4):255–299, Nov 1990.
26. Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3), September 2001.
27. A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R Motwani, U. Srivastava, and J. Widom. STREAM: The Stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004.
28. César Sánchez. Online and offline stream runtime verification of synchronous systems. In *Proceedings of the 18th Int'l Conf. on Runtime Verification (RV'18)*, volume 11237 of *LNCS*, pages 138–163. Springer, Berlin, 2018.
29. Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David Rydeheard. Quantified event automata: towards expressive and efficient runtime monitors. In *Proceedings of the 18th Int'l Symp. on Formal Methods (FM'12)*, volume 7436 of *LNCS*, pages 68–84. Springer, Berlin, 2012.
30. David A. Basin, Felix Klaedtke, Samuel Müller, and Eugen Zalinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(1), 2015.
31. Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. A stream-based specification language for network monitoring. In *Proceedings of the 16th Int'l Conf. on Runtime Verification (RV'16)*, volume 10012 of *LNCS*, pages 152–168. Springer, Berlin, 2016.