Stream-based Monitors for Real-time Properties[⋆]

Hazem Torfah

Reactive Systems Group
Saarland University Saarbrcken, Germany
torfah@react.uni-saarland.de

Abstract. In stream-based runtime monitoring, streams of data, called input streams, which involve data collected from the system at runtime, are translated into new streams of data, called output streams, which define statistical measures and verdicts on the system based on the input data. The advantage of this setup is an easy-to-use and modular way for specifying monitors with rich verdicts, provided with formal guarantees on the complexity of the monitor.
In this tutorial, we give an overview of the different classes of stream specification languages, in particular those with real-time features. With the help of the real-time stream specification language RTLola, we illustrate which features are necessary for the definition of the various types of real-time properties and we discuss how these features need to be implemented in order to guarantee memory efficient and reliable monitors. To demonstrate the expressive power of the different classes of stream specification languages and the complexity of the different features, we use a series of examples based on our experience with monitoring problems from the areas of unmanned aerial systems and telecommunication networks.

Keywords: Stream-based Monitoring · Real-time Properties · Stream Specification Languages.

1 Introduction

The online monitoring of real-time data streams has recently gained a great deal of attention, especially with the growing levels of autonomy and connectivity in modern real-time systems [1, 7–9, 14, 17, 28, 30]. Runtime monitors are essential for evaluating the performance and for assessing the health of a running system, and are integral for the detection of malfunctions and consequently for deploying the necessary counter measures when these malfunctions occur at runtime.

⋆ This work was partially supported by the German Research Foundation (DFG) as part of the Collaborative Research Center "Foundations of Perspicuous Software Systems" (TRR 248, 389792660), and by the European Research Council (ERC) Grant OSARES (No. 683300).

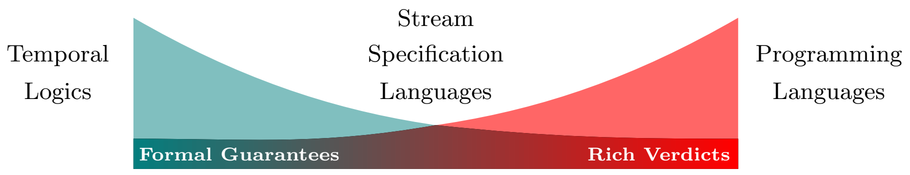


Fig. 1: Spectrum of specification languages and tradeoffs in implementing monitors for real-time properties.

The integration of monitoring components into real-time systems demands the construction of monitors that are (1) *efficient*: using an on-board monitor that consumes a large amount of memory is bound to eventually disrupt the normal operation of the system, and (2) *reliable*: in the case of any malfunction or abnormality in one of the system's components, the monitor needs to provide, as early as possible, a correct assessment for deploying the right fallback procedure.

Monitors for real-time properties can be defined using a variety of languages, ranging over a spectrum that spans from formal languages with strong guarantees, such as temporal logics (e.g. [2, 25, 29]), to very expressive languages that allow for the definition of monitors with rich verdicts, such as scripting and programming languages. Moving from one end of the spectrum to the other, there is always a trade-off between the expressivity of a language and the guarantees it provides on the constructed monitors (Figure 1). Monitors specified in temporal logics come with formal complexity guarantees, involving bounds on the maximum memory consumption of the monitor at runtime. Furthermore, temporal logics have the big advantage of allowing for the automatic construction of monitors from their specifications, and the guarantee that these monitors fulfill these specifications. A disadvantage of temporal logics is, however, that they are limited to describing monitors with simple boolean verdicts, which do not always suffice for the definition of monitors for practical real-world monitoring problems, where more involved computations over more complex datatypes are needed. Programming languages on the other hand allow for the implementation of such complex computations. This comes at the cost of losing formal guarantees on the constructed monitors, and also on their reliability, as there is no guarantee on the soundness of the manually implemented monitor.

In general, different monitoring applications require different approaches to constructing monitors. In modern real-time systems, monitors are required to perform complex arithmetic computations, but at the same time perform these computations with respect to the limited resources available on the platform they are running on. To implement such monitors for real-time properties, we need specification languages that are expressive enough, and, at the same time, provide certain guarantees on the complexity of the constructed monitor.

In this tutorial, we show how stream specification languages with real-time features can be used for the specification of runtime monitors for real-time properties. Stream specification languages allow for the definition of stream-based monitors, where input streams, containing data collected at runtime, such as sensor readings, or network traffic, are translated into output streams, containing aggregate statistics, threshold assertions and other logical conditions. The advantage of this setup is that it combines features from both sides of the specification language spectrum: the great expressiveness of programming languages on one hand, and the ability to compute a-priori formal guarantees on the monitor, as in the case for temporal logics, on the other hand.

We give an overview of the different types of real-time stream specification languages and illustrate, using the stream specification language RTLola [14, 15], the different features needed for expressing monitors for real-time properties. We further show how these features need to be implemented in order to obtain memory efficient and reliable monitors. To demonstrate the expressive power of stream specification languages in general and RTLola in particular, we will rely on examples from real-world monitoring problems from the area of unmanned aerial vehicles, which we have been investigating in close collaboration with the German Aerospace Center (DLR) [1], and from our experience with problems from the field of network monitoring [13].

This tutorial is structured as follows. In Section 2, we give an introduction to stream specification languages and show the different classes of these languages, according to their underlying computational model (synchronous vs. asynchronous), and the ways streams are accessed in the language (discrete-time vs. real-time access). In Section 3, we show the advantage of adding parameterization to stream specification languages and briefly explain the challenges in monitoring parameterized stream specifications. Section 4 shows how real-time stream specification languages subsume real-time logics like STL [25], and the role of parameterization in encoding STL formulas in stream specification languages like RTLola. In Section 5, we give an overview of the various monitoring approaches for real-time properties. Finally, in Section 6, we conclude our survey with a brief discussion on the usage of stream-based languages in practice, and mention some works that have been done along this line.

2 Stream Specification Languages

A stream-based specification defines a runtime monitor by describing the relation between the values of input streams, entering the monitor, and the values of the output streams, computed by the monitor. For example, if an input stream contains the elevation values measured by an altimeter in some aerial vehicle, the monitor could compute the output stream that determines whether the measured elevations are below a certain altitude. In a telecommunication network, we may deploy a monitor that checks the frequency at which data is received on a node in the network over a period of time. The input stream to this monitor is the stream of data packets entering the node, and the output stream of the monitor is a stream of values where each value defines the number of packets that entered the network, for example, in the last second.

Stream specification languages are classified according to the type of monitors they can describe. The distinction depends, in general, on *which* values are allowed to be used in the definition of the output stream, and *when* the values of an output stream are computed. With respect to what values can be used in the definition of streams, we can distinguish between languages that allow for the definition of *rule-based* monitors, or *memoryless* monitors, i.e., monitors which do not need to memorize any previous stream values to compute the new values, and languages that allow for the definition of *state-based* monitors, i.e., monitors that can maintain a memory of previous values that were computed by the monitor. In state-based monitors, we further distinguish between how previous values of streams are accessed in the stream definition. Stream accesses can be either in *discrete-time*, i.e., in the definition of the output stream, concrete previous values of other streams were accessed. A stream access can also be a *real-time* access, i.e., the output stream has access to the values of another stream that occurred over a certain period of time.

In addition to maintaining the necessary memory of values for the computation of output streams, we also have to specify when the values of an output stream are to be computed. Values of an output stream can, for example, be computed every time the monitor receives a new input value, or at certain fixed points or periods of time. The computation models for stream-based monitors can be categorized according to when input data arrives into the monitor and whether the computation of output values depends on the arrival times of input data. In general, we can distinguish between the *synchronous* and the *asynchronous* computation models. In the synchronous model, new data on different input streams arrive at the same time, and the values of output streams are computed with every arrival of new input data. A prominent example of monitoring problems with a synchronous computation model are network monitoring problems where the monitoring task is defined over the individual packets arriving to a node in the network. In the asynchronous computation model, input data on different input streams may arrive at different times and the values of the output streams may be computed independently of the arrival of input values. Such a computation model can be found in any cyber-physical system with different sensors that function at different rates.

In the following we will elaborate on each of these notions with the help the stream specification language RTLola [14]. RTLola is a state-based asynchronous stream specification language with real-time features that allows for the definition of monitors for real-time properties over rich datatypes. An RTLola specification defines a typed set of output streams given as a set of typed stream equations that map output stream names to expressions over input streams and other streams defined by the specification[1].

[1] For the complete syntax of RTLola we refer the reader to www.stream-lab.org

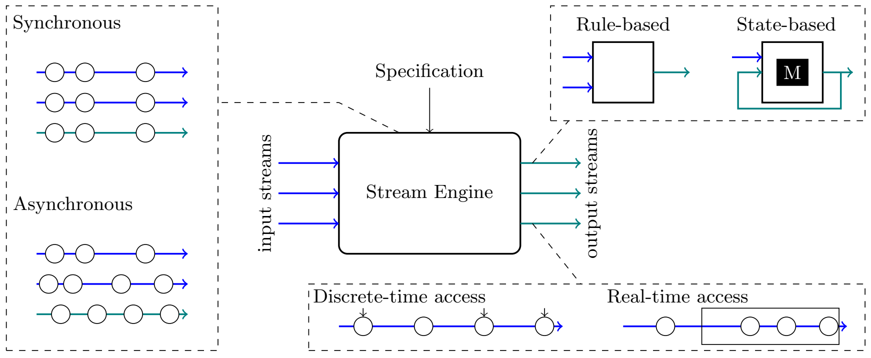


Fig. 2: Classification of stream specification lanaguages.

2.1 A Classification of Stream Specification Languages

Rule-based vs. State-based Specifications. We distinguish between two types of stream specification languages with respect to whether the computation of an output stream depends on previously computed values. *Rule-based* languages are those specification languages that allow for the definition of memoryless monitors, i.e., computing the value of an output stream depends only on the current values of input streams and are independent from any history of values that have been computed up to that moment[2]. Monitors specified in a rule-based stream specification language are also known as stateless monitors and are very common in network intrusion detection [19, 24].

An example of a rule-based stream-based monitor is given by the RTLola specification

```
input packet: IPv4
output empty_UDP: Bool := packet.isUDP() & packet.isEmpty()
```

Every time a data packet is received on the input stream `packet`, which represents the stream of packet traffic in some node in a network, a new event is computed for the output stream `empty_UDP`. If the packet is a UDP packet and has an empty payload, then the newly computed value of `empty_UDP` is true, otherwise it is false.

Another example of a rule-based stream definition is given by the following specification. Assume we want to monitor the vertical geofencing boundaries of a flying drone. A monitor for this task can be defined by the specification

```
input altimeter: UInt32
output too_low: Bool := altimeter < 100
```

[2] The term memoryless here does not consider the memory needed to perform an operation on the current values of input streams in order to compute the value of the output stream, but refers the number of previous values that need to be stored to compute the current output value.

For each value received from the altimeter, a new output value for the stream `too_low` is computed that registers whether the drone is flying below 100 feet.

Rule-based specification languages are easy to use and allow for the construction of simple and efficient monitors. They are, however, not suited for specifying complex monitoring tasks that need to maintain a state.

Consider for example a monitor that extends the vertical geofencing monitor by additionally counting the number of times the drone flew below the allowed height. A stream specification for such a monitor looks as follows

```
output count: UInt32 := if too_low
                         then count.offset(by: -1) + 1
                         else count.offset(by: -1)
```

The output stream represents a counter that is increased every time a new value is received from the altimeter, and when that value is below 100 feet. The new value of the stream `count` thus depends on its last computed value, which is defined in the RTLola specification by the expression `count.offset(-1)`. To implement a monitor for this specification, the monitor needs to store the last value of the output stream in order to compute its new value.

Monitors that need to maintain a state can be defined using *state-based* specification languages like RTLola. State-based stream specification languages are powerful languages that combine the ease-of-use of rule-based specification languages with the expressive power of programming languages, when equipped with the right datatypes. State-based languages are common in state-based intrusion detection [12, 27], or in developing mission control and flight managing tasks in UAVs [1].

Discrete-time vs. Real-time Stream Access. In state-based approaches a monitor may compute a value of an output stream depending on a history of values received on or computed over other streams, including the own history of the output stream. In the following we distinguish between the different types of stream access.

Accessing the values of other streams can be done in a discrete manner, i.e., the value of an output stream depends on certain previous values of some other stream. For example, in the stream `count`, the value of the stream depends on its own lastly computed value. A stream may also depend on the last n values of a stream, for example, when we want to compute a discrete sliding window over events. If we want to check whether the drone has been flying below the minimum altitude during the last three values received from the altimeter, we can define such a monitor using the following specification

```
output last_three: Bool :=
                 too_low
               & too_low.offset(by: -1).defaults(to: false)
               & too_low.offset(by: -2).defaults(to: false)
```

Here, we access the current and, using the offset operator, the last and the before the last value of the stream `too_low`.

When using offset expressions to access previous values of a stream, we need to make sure that such values actually exist. A new value for the stream `last_three` is computed whenever a new value is computed for `too_low`. At the beginning of the monitoring process, the monitor may not have computed three values for `too_low` yet, because we have not received the necessary number of readings from the altimeter. Assume that we have received the first value over the input `altimeter`. Then, the values `too_low.offset(-1)` and `too_low.offset (-2)` are not yet defined. To still be able to evaluate the stream `last_three`, we need to divert to a fixed default value given in the specification above by the expression `defaults(to: false)`. This expression returns the value false when the stream expressions `too_low.offset(-1)` or `too_low.offset(-2)` are not defined.

Discrete offset expressions can also be used to access future values of streams. Consider for example a monitor that checks whether a drone reaches a specific waypoint with GPS coordinates (a,b), a geographic location the drone is commanded to fly to. A specification for such monitor is given as follows

```
input gps: (Float64,Float64)      // (latitude, longitude)
output reached_wp: Bool :=
               gps == (a,b) |
               reached_wp.offset(by: 1).defaults(to:false)
```

The monitor checks whether the current GPS coordinates match the targeted waypoint and if not repeats the check for the next received GPS value. In general, most of the monitors defined by stream specifications with future offsets can be defined with stream specifications with only past offsets. The future offsets are nevertheless convenient for encoding specifications given as formulas in temporal logics. One has to be careful however in using future offsets, as they introduce the possibility of defining ill-formed specifications, when they contain zero circular offset dependencies [9, 33].

An output stream may also depend on the values that occurred in a certain time interval. Real-time stream access can be achieved using real-time sliding windows, where an aggregation of events is computed over a fixed period of time. Consider for example a monitor that checks whether packets are received in large bursts. The specification of such a monitor can be given by the RTLola specification

```
input packet: IPv4
output burst: Bool :=
     packet.aggregate(over: 10sec, using: count)
     > threshold
```

For each new event received on the input stream `packet`, the monitor checks whether more than `threshold` many packets have been received over the input stream in the last 10 seconds.

Sliding windows can be implemented in two versions, *forward* and *backward* sliding windows. A backward sliding window of duration d is interpreted as in the last specification where the aggregate is defined over the values of the interval of d seconds until the last value arrived. Forward sliding windows consider the values in the period of length d starting with the time at which the last value arrived. Forward sliding windows are necessary for the specification of monitors that check timeouts. For example, to check whether the drone reaches a certain height `h` within `d` seconds we can specify the following monitor

```
input gps: (Float64,Float64)
input coordinate: (Float64,Float64)

output too_slow: Bool :=
    gps.aggregate(over: +2min, using: exists(coordinates))
```

where `exists(coordinates)` is syntactic sugar for an aggregation function that checks whether we received a value on the stream `coordinates` during a window of two minutes. Backward windows cannot be used to specify such monitoring tasks, at least not accurately, as they might miss some values (we explain this below). Backward sliding windows nevertheless come with the advantage that their computation does not have to be delayed as for the case of forward sliding windows, because they only rely on the events that have already been observed by the monitor. In general, one may use backward sliding windows to perform timeout checks, if the computational model of the specification language also allows to evaluate output streams at certain rates, as we will see later. This may come however at the cost of missing some windows, if the granularity in which the stream is computed is not small enough. We explain this with the help of the following example. Assume we want to check that there are no windows with a duration of one second that have events with values less than 3. A specification for such a monitor looks as follows

```
input x: UInt32
output y: UInt32 :=
             if x.aggregate(over: 1sec, using: max) <= 2
              then y.offset(by: -1).defaults(to: 0) + 1
              else y.offset(by: -1).defaults(to: 0)
```

Assume that we receive events 5,4,2,2 on the stream `x` at times 0.1, 0.2, 0.6, and 1.4 seconds. A backward window approach will count 1 window with a maximum number of 2, namely when evaluating the stream at the arrival of the fourth event. A forward approach will count 2 windows when evaluated for the third an the fourth event. With the right rate, nevertheless, we can count the right number of windows, for example, if the output stream is computed every 0.5 seconds.

Synchronous vs. asynchronous computation models. The evaluation of an output stream may depend on the values of several input streams. The computation of an output stream may thus depend on the arrival times of the input values. In general, we distinguish two computation models, the *synchronous* model and the *asynchronous* model.

In the synchronous model, events on all input streams arrive to the monitor at the same time. Output streams are evaluated with the arrival of new inputs values and thus are also evaluated simultaneously. Examples of systems with synchronous models are synchronous circuits and networks [9, 13].

In the asynchronous model, the arrival times of inputs may vary from one input stream to another, and the computation of output streams does not necessarily depend on the arrival times of input values. In a cyber-physical system, for example, sensors may function at different frequencies, and thus data from these sensors arrive at different not necessarily synchronized rates. Computing the values of output streams may respect the arrival times of input values, or could completely be decoupled from them. In general, we distinguish two types of output streams with respect to their dependency on the rate in which input values reach the monitor, namely, *time-triggered* and *event-triggered* streams.

Time-triggered streams. In time-triggered output streams, values are computed at determined times, which are independent of the arrival of input values. These determined times can be periodic or dynamically determined. In periodic output streams, a new output is computed at a fixed frequency. Periodic monitors are associated with tasks for validating sensor frequencies in cyber-physical systems. For example, a monitor for checking whether a GPS sensor is delivering data in the right frequency can be specified as follows

```
input gps: (Float64,Float64)

output gps_glitch: UInt32 @ 1Hz:=
            gps.aggregate(over: 2sec, using: count)

trigger gps_glitch < 10 "GPS sensor: frequency < 5Hz"
```

The stream `gps` is an input stream that represents the values received from the GPS module and is expected to deliver data with a frequency greater than or equal to `5Hz`. To check whether this data is delivered with the expected frequency, we define the output stream `gps_glitch`. A new value for the output stream `gps_glitch` is computed every second (the expression `@ 1Hz`), and at each time, it evaluates to the number of values received from the GPS module over a time window of two seconds, i.e., it computes the number of events received via the input stream `gps` (the expression `using: count`) over the last two seconds (the expression `over: 2sec`). The stream `trigger gps_glitch < 10` defines an assertion that evaluates to true when the value of `gps_glitch` is less than 10 (`trigger` is a special keyword adapted from the language Lola [9] that raises an alarm and outputs a message in case the assertion is true).

Dynamically evaluated output streams allow further to use a function defined over streams to determine when to compute the next value of an output stream. An example of such a function is one that allows for the specification of streams where the output stream is delayed 5 seconds after `gps` received a new value[3]

[3] The version of RTLola that is currently implemented in StreamLAB [14] does not allow for activation conditions with delay, but the implementation of such conditions is planned for the near future. Striver [17] and TeSSLa [8] have a native delay operator.

```
output gps_glitch: UInt32 @ gps + 5 :=
            gps.aggregate(over: 2sec, using: count)
```

Event-triggered streams. In an event-triggered output stream, a new value is computed for this stream depending on the arrival times of new input events. An example of an event-triggered output stream is one where we extend our GPS specification with a definition of an output stream that counts the number of glitches observed

```
output num_glitches: UInt32 :=
   if gps_glitch
    then num_glitched.offset(by: -1).defaults(to:0) + 1
    else num_glitches.offset(by: -1).defaults(to:0)
```

The specification defines a monitor that, with each new value computed for the stream `gps_glitch`, computes the number of glitches seen so far. This means that the pace in which `num_glitches` is computed is equal to the pace of `gps_glitch`.

If the output stream is defined over more than one input stream, then the new value of the output stream can be computed in different ways. In some cases it makes sense to use a hold semantics, where with each arrival of a new event on some input stream, the value of the output stream is computed with the latest values of all input streams. This is typical for monitors over piecewise constant signals. Piecewise constant signals can be represented as discrete signals where every change in the signal is a new event. In the continuous world, an operation over two piecewise constant signals result in a new piecewise constant signal that in each point evaluates to the operation on the values of the two signals at that point. For example, if we have two signals $a : \mathbb{R}_{\geq 0} \to \mathbb{R}$ and $b : \mathbb{R}_{\geq 0} \to \mathbb{R}$, then the signal representing the sum of a and b, is a signal $s(t) = a(t)+b(t)$. To monitor whether the sum exceeds any threshold we construct the signal s and check if the value of s is larger than the threshold. Over the discrete representation, a monitor is given by the specification

```
input a: UInt32, b : UInt32
output exceededThreshold: Bool @ (a|b)
:= a.hold() + b.hold() > c
```

This output stream is evaluated every time `a` or `b` receives a new value. If `a` has a new value and `b` does not, then the value of the output stream is computed using the new value of `a` and the last received value of `b`. This is determined by the use of the access via the zero-order hold operator `hold()` which returns the last computed value of a stream. The condition `a|b` is called the activation condition. If the activation condition is empty then the stream is only computed if all values arrive at the same time.

Another advantage of the activation condition and the hold semantics is that in the case where the values of sensors arrive with slight delays, we can use the activation condition to evaluate the stream when the later value arrives and the hold operator to use the last value of the other sensor. Consider the following specification

```
input gps: (Float64,Float64)
input height: Float64
output too_low:Bool @ gps := if zone(gps)
  then (height.hold().defaults(to: 300)) < 300
  else false
```

where the function `zone` is some function that determines whether the drone is in an inhabited area. The output stream is evaluated every time the GPS sensor delivers a new value. The value of the stream `too_low` is determined using the last value of `height`.

2.2 Memory Analysis

To compute memory bounds on a monitor specified in RTLola, we need to analyze the stream accesses in the monitor's specification. For output streams that only use offset expressions to access other streams, memory bounds can be computed in the same way as for Lola [9, 33]. To give an idea on how these bounds are computed, we describe the process using the following examples.

Consider again the RTLola specification

```
output last_three: Bool :=
                too_low.offset
              & too_low.offset(by: -1).defaults(to: false)
              & too_low.offset(by: -2).defaults(to: false)
```

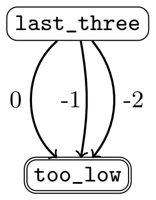
To evaluate an event of the output event `last_three`, we need the last three values of the streams `too_low`. This means that the monitor requires two memory units, where the last and the value before the last are saved, in order to be able to compute the values for the stream `last_three`.

Consider further the specification

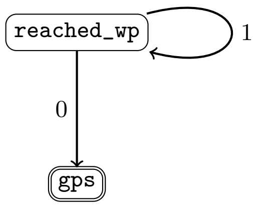
```
input gps: (Float64,Float64)      // (latitude, longitude)
output reached_wp: bool :=
               gps == (a,b) |
               reached_wp.offset(by: 1).defaults(to:false)
```

To compute a value for the output stream `reached_wp`, the monitor checks whether the drone has reached the designated waypoint `(a,b)`, otherwise it will wait for the next values received from the GPS sensor to compute the current value of `reach_wp`. If the waypoint `(a,b)` is not reached, computing a value for `reached_wp` remains pending. This means that the monitor needs to save all unresolved values for `reached_wp` until the coordinates `(a,b)` are reached. In the hypothetical case that the drone will never reach these coordinates, the monitor needs to memorize an infinite number of unresolved values for `reached_wp`, which results in problems when the monitor is only offered a limited amount of memory to compute its streams.

The memory bounds can be automatically computed by constructing the annotated dependency graph of a specification [9]. The dependency graphs for the specifications above are depicted in Figure 3. A RTLola specification is



(a) An annotated dependency graph for the stream `last_three`. A monitor for `last_three` needs to save the last two events of `too_low`.



(b) An annotated dependency graph for the stream `reached_wp`. A monitor for `reached_wp` needs to save a possibly infinite number of unresolved expressions of itself.

Fig. 3: Annotated dependency graphs for the RTLola specifications of the streams `last_three` and `reached_wp`.

called efficiently computable, if we can determine memory bounds on the monitor specified by the specification. This is the case when the dependency graph of a specification does not contain any positive cycles. If an RTLola specification is efficiently computable, then we can compute the memory bounds for a stream by computing the maximum of the maximum added positive weights on a path starting from the stream's node, or the maximum absolute value of the negative weights on edges exiting the node.

For specifications with sliding windows, and in the case of input streams with variable arrival rates, it is in general not possible to compute memory bounds on the monitors implementing these specifications. The reason for this lies in the fact that the number of events that may occur in a period of time might be arbitrary large. Nevertheless, in most cases the rates in which input data arrives on a an input stream are bounded by some frequency. If we do not know this frequency, we can still compute sliding windows efficiently by using periodic time-triggered streams. We use the following example to demonstrate how computing aggregates over sliding windows can be done with bounded memory[4]. Consider the specification

```
input packet: IPv4
output burst: Bool @ 1Hz:=
     packet.aggregate(over: 5 sec, using: count)
```

Instead of counting the number of packets that arrive within a window every time the sliding window needs to be evaluated, we split the window into intervals, called panes, and count the number of events in each pane [23]. When we compute the aggregate for a new window, we reuse the values of the panes that overlap with the new window to compute the new aggregate. Figure 4 illustrates the computation of the sliding window for our specification above. The memory needed for computing the aggregate count over the sliding window is bounded

[4] For more on the implementation of sliding windows we refer the reader to [5, 14, 23]

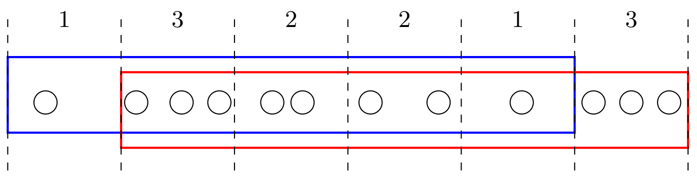


Fig. 4: Efficient computation of the aggregate count over a sliding window over 10 seconds that is computed in periods of one second. The value of the new window is equal to 9-1+3, where 9 is the value of the aggregation over the last window.

by five memory units that save the aggregate for each of the five one-second panes. Each second the window moves by one second to the right. To compute the aggregate over the new window, we just need to remove the value of the first pane form the value of the last window and add the value of a new pane that includes all events that arrived within the last second. The first pane of the last window can be removed from memory and the value of the new pane is saved instead.

Splitting windows into panes showed to be very useful in practice [7, 14]. However, it can only be applied to aggregation functions that fulfill certain properties [14]. The approach would for example not work for aggregation functions such as the median. Nevertheless, from our experience most aggregation functions used in practice can be efficiently solved by this approach. These include aggregation function such as count, sum, max, min and integral.

3 Parameterized Stream Specifications

In many cases, the same monitoring task has to be performed for multiple sets of data. For example, in an online platform with a large user base, we might want to compute the same statistics over the different user groups of the platform, distinguished by age, gender, or geographic location. To compute the statistic for each group we can define an output stream for each user group, but this results in unnecessary redundant stream definitions. The redundancy can be avoided by defining one stream using appropriate datatypes such as sets, maps, etc. This, however, may result in incomprehensible specifications and cumbersome extra work for managing the complex datatypes. Furthermore, when the statistics need to be computed separately for each individual user, writing different stream definitions for each user is not feasible and managing the computation in one stream results in a large overhead caused by updating the complex datatypes.

To overcome this problem, we can define parameterized monitoring templates, from which new monitors can be instantiated every time we want to perform the monitoring task for a specific user. Parameterized stream specifications allow for the dynamic construction of streams, when these streams are needed, and that run on their own individual pace.

Consider, for example, the following parameterized RTLola specification for monitoring the log-in's of users to some online platform

```
input user_activity: UInt32

output timeout(id: UInt32): Bool @ 1min
close timeout(id)
:=
user_activity.aggregate(over: 10min, using: count(id)) == 0

output logout: UInt32 @ user_activity :=
              if timeout(user_activity).hold()
               then user_activity
               else -1
```

The specification defines an output stream template `timeout` with one parameter `id` of type `UInt32`. An instance of the template `timeout` is a stream `timeout(x)` for a concrete value `x` of type `UInt32`. At the beginning, there are no instances for the template `timeout`, and instances for a concrete value `x` are only created when they are called by the stream `logout`. Every time a new user id is observed on the input stream `user_activity`, the stream `logout` calls the value of the instance `timeout(user_activity)`. If the instance has not been created yet, then a new instance with the current value of `user_activity` is created and evaluated according to the expression of `timeout` with respect to the new parameter value. Here, the new instance evaluates to true, if the user with the id `user_activity` has not been active in the last 10 minutes. If the instance already exists, then `logout` is evaluated according to the last value computed for the instance. If a user with id `x` has not been active for more than 10 minutes, the instance `timeout(x)` is terminated, as defined by the termination expression `close: timeout(id)`.

Memory bounds for parameterized stream specifications can be computed in the same way as for non-parameterized ones, with the significant difference, that the memory bounds determine the size of memory needed to compute each instance. The number of instances created by the monitor, and the number of instances that are active simultaneously depends highly on the application. In general, the number is limited, as most created streams define simple monitoring tasks that reach a verdict quickly. In the case, where the monitor is forced to produce a large number of instances at once, it is recommended to force the termination of instances after a period of time.

4 Embedding Real-time Logics in RTLola Using Parameterized Specifications

Stream specification languages like Lola that provide operators for referencing values in the future subsume temporal logics like LTL [9]. LTL is a linear-time logic that allows us to define properties over traces using boolean and temporal operators. Temporal operators in LTL are used to reason about time and are given by the operator "next" ($\bigcirc\varphi$), that states that a certain property φ must hold in the next step, and the operator "until" ($\varphi\,\mathcal{U}\,\psi$) that states that a property φ must hold until another property ψ is satisfied. The temporal operators can be specified in RTLola by the following stream specifications

```
output next: Bool := phi.offset(1).defaults(to: true)

output until: Bool :=
           psi |
          (phi & until.offset(1).defaults(to: false))
```

where `phi` and `psi` are stream definitions for the formulas φ and ψ.

For real-time logics like STL [25], the offset operator does not suffice as we need to check the values of signals at certain points in time. In the following we show how we can use parameterized RTLola specifications to encode STL specifications over piece-wise constant signals. The next definition gives a short recap on the syntax of STL.

Definition 1 (Signal Temporal Logic [25]). *An STL formula φ over signals $x_1, \ldots, x_m \in \mathbb{R}_{\geq 0} \to \mathbb{R}$ is given by the grammar*

$$\varphi = \mu \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi\,\mathcal{U}_{[a,b]}\,\varphi$$

where μ is a predicate from $\mathbb{R} \to \mathbb{B}$ with $\mu(t) = f(x_1[t], \ldots, x_m[t]) > 0$ for some function $f : \mathbb{R}^m \to \mathbb{R}$.

Given piecewise-constant signals $x_1, ..., x_m$, an STL formula φ, and predicates $\mu_1, \ldots, \mu_n$ we can encode the monitoring problem for φ in RTLola using the following recursively defined translations:

– x_i for $1 \leq x \leq m$:

```
input x_i: Float64
```

where `x_i` receives a new event every time the signal x_i changes its value.

– μ_j for $1 \leq j \leq n$:

```
output mu_j(t: Time): Bool @ any :=
                 f(x_1.hold(),...,x_m.hold())>0
```

where `f` is an operation that defines the value of f. The stream template `mu_j` defines streams that represents the values of the expression `f(x_1,...,x_m)` starting at time `t`. Once a stream has be created, it is evaluated every time one of the streams `x_i` receives a new value (abbreviated by the word `any`).

– $\neg\varphi$:

```
output nphi(t: Time): Bool := !phi(t)
```

The stream computes the negated values of the stream `phi(t)`, and is evaluated at the pace of `phi`.

– $\varphi_1 \vee \varphi_2$:

```
output orphi1phi2(t: Time): Bool @ any
:= phi1(t).hold() | phi2(t).hold()
```

The stream computes the disjunction of the streams `phi1` and `phi2`.

– $\varphi_1 \mathcal{U}_{[a,b]} \varphi_2$. For such a formula, we need to check whether the formula φ_2 is true at some point $t \in [a, b]$, and the formula φ_1 must hold up to the point t. If φ_2 is also an until formula defined for some interval $[c, d]$ then the validity of φ_2 must then be checked relative to the time t. This is encoded in RTLola as follows.

```
output untilphi1phi2(t: Time) : Bool @ (t+b)| any
close: time == b | !untilphi1phi2(t)
:=
if time <= t+a
 then
   phi1<time>.hold() & untilphi1phi2[a,b](t).offset(1)
 else
  if time < t+b
   then
    phi1(time).hold() &
    (phi2(time).hold() |
      untilphi1phi2[a,b](t).offset(1))
   else
    phi1(time).hold() & phi2(time).hold()

trigger untilphi1phi2[a,b](0)
```

The complete STL formula must hold at time 0. Therefore we check the value of the formula by calling the stream `untilphi1phi2[a,b](0)`. An instance `untilphi1phi2(x)` is evaluated whenever a value of one of its substreams is computed or when reaching the time mark $x + b$. The evaluation at $x + b$ is necessary for the case that none of the signals change their values up to that point. At every evaluation point of `untilphi1phi2(x)`, we check in which time interval we are. If the current `time` is smaller than $x + a$, then we check that the value of `phi1(x)` is true. If this is the case, we check the validity of the right formula in the designated future interval $[a, b]$ by calling `unitlphi1phi2 [a,b].offset(1)`. If we arrive at the time interval $[x+a, x+b]$ we check again whether the left formula still holds, and whether the right formula holds. If we exceed the time $x + b$ and the right formula has not been true yet then the stream is evaluated to false.
At each time t, where the instance is evaluated, new instances with parameter instantiation t are created for the subformulas. All instances are terminated once their intervals are exceeded.

Remark 1. Formulas in STL can also be encoded without the usage of parameterized stream definition, if the stream specification language allows for the definition of activation conditions where the evaluation of a stream can be set to certain points in time. Such activation conditions can be defined in languages like Striver [17].

5 Bibliographic Remarks

Most of the early work on formal runtime monitoring was based on temporal logics [11, 16, 18, 21]. The approaches vary between inline methods that realize a formal specification as assertions added to the code to be monitored [18], or outline approaches that separate the implementation of the monitor from the one of the system under investigation [16]. Based on these approaches and with the rise of real-time temporal logics such as MTL [20] and STL [25], a series of works introduced new algorithms and tools for the monitoring of real-time properties described in one of the previous logics [3, 4, 6, 10, 26, 32].

The stream-based approach to monitoring was pioneered by the specification language Lola [9, 33], a descriptive language, that can express both past and future properties. The main feature of Lola is that upper bounds on the memory required for monitoring can be computed statically. RTLola extends Lola with real-time features, such as sliding windows and time-triggered stream definitions, and allows for the definition of monitors with an asynchronous computation model. RTLola adapts the memory analysis techniques provided by Lola, and extends those techniques for determining memory bounds for the new real-time features added to the language.

Further real-time stream specification languages based on Lola were introduced with the languages TeSSLa and Striver. TeSSLa [22] allows for monitoring piece-wise constant signals where streams can emit events at different speeds with arbitrary latencies. TeSSLa has a native delay operator that allows for the definition of future times in which streams shall be computed. The version of RTLola that is currently implemented in the tool StreamLAB [14] does not yet have such a delay operator. On the other hand, TeSSLa does not have native support for real-time features such as the definition of sliding windows. Striver, is a stream specification language that allows the definition of involved activation conditions, that especially allow for the definition of timeout specifications. Using a tagging mechanism and the delay operator in Striver, one can define forward sliding windows. There is however no native operator in the language for the definition of sliding windows, for which built-in efficient algorithms are implemented, as in the case of RTLola.

Further stream-based specification languages with real-time features include the languages Copilot [28], which allow for the definition of monitors based on the synchronous computation model. Prominent specification languages specialized for specifying monitors for network intrusion detection include the frameworks Bro[27] and snort [31].

6 Conclusion

In this tutorial, we gave an overview of the different classes of stream specification languages for specifying monitors for real-time properties, and demonstrated, with the help of the stream specification languages RTLola, which features allow for the definition of monitors for the different types of real-time properties. We further discussed the construction of memory efficient monitors and showed how memory bounds can be computed for monitors written in the stream specification language RTLola. Finally we discussed parametric extensions to stream specification languages and showed that real-time logics such as STL are subsumed by RTLola with parameterization.

From our experience, stream specification languages like RTLola are well received by practitioners [1, 34, 35]. The outline monitoring approach given by stream-based monitors allows for the separation between the monitoring component and the system under scrutiny, which has the advantage of not interfering with the functionality of the system. Furthermore, stream specification languages provide a specification framework that is simple to use, easy to understand and one that combines features of high expressive scripting languages that are used in industry with important features of formal languages such as computing memory bounds.

Acknowledgements. I would like to thank Bernd Finkbeiner, Norine Coenen, Christopher Hahn, Maximilian Schwenger and Leander Tentrup for their valuable feedback and comments.

References

1. Adolf, F.M., Faymonville, P., Finkbeiner, B., Schirmer, S., Torens, C.: Stream runtime monitoring on UAS. In: Lahiri, S., Reger, G. (eds.) Runtime Verification. pp. 33–49. Springer International Publishing, Cham (2017)
2. Alur, R., Henzinger, T.A.: Logics and models of real time: A survey. In: Proceedings of the Real-Time: Theory in Practice, REX Workshop. pp. 74–106. Springer-Verlag, Berlin, Heidelberg (1992), http://dl.acm.org/citation.cfm?id=648143.749966
3. Basin, D., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (May 2015). https://doi.org/10.1145/2699444
4. Basin, D.A., Harvan, M., Klaedtke, F., Zalinescu, E.: MONPOLY: monitoring usage-control policies. In: Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011. pp. 360–364 (2011). https://doi.org/10.1007/978-3-642-29860-8_27
5. Basin, D.A., Klaedtke, F., Marinovic, S., Zalinescu, E.: Monitoring of temporal first-order properties with aggregations. Formal Methods in System Design **46**(3), 262–285 (2015). https://doi.org/10.1007/s10703-015-0222-7
6. Basin, D.A., Krstic, S., Traytel, D.: AERIAL: Almost event-rate independent algorithms for monitoring metric regular properties. In: RV-CuBES. Kalpa Publications in Computing, vol. 3, pp. 29–36. EasyChair (2017)

7. Baumeister, J., Finkbeiner, B., Schwenger, M., Torfah, H.: FPGA stream-monitoring of real-time properties. In: ESWEEK-TECS special issue, International Conference on Embedded Software EMSOFT 2019, New York, USA, October 13 - 18 (2019)
8. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: Tessla: Temporal stream-based specification language. In: Brazilian Symposium on Formal Methods. Lecture Notes in Computer Science, vol. 11254. Springer (2018)
9. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: Lola: Runtime monitoring of synchronous systems. In: 12th International Symposium on Temporal Representation and Reasoning (TIME'05). pp. 166–174. IEEE Computer Society Press (June 2005)
10. Deshmukh, J.V., Donzé, A., Ghosh, S., Jin, X., Juniwal, G., Seshia, S.A.: Robust online monitoring of signal temporal logic. In: Bartocci, E., Majumdar, R. (eds.) Runtime Verification. pp. 55–70. Springer International Publishing, Cham (2015)
11. Drusinsky, D.: The temporal rover and the atg rover. In: Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification. pp. 323–330. Springer-Verlag, London, UK, UK (2000), http://dl.acm.org/citation.cfm?id=645880.672089
12. Eckmann, S.T., Vigna, G., Kemmerer, R.A.: Statl: An attack language for state-based intrusion detection. J. Comput. Secur. **10**(1-2), 71–103 (Jul 2002), http://dl.acm.org/citation.cfm?id=597917.597921
13. Faymonville, P., Finkbeiner, B., Schirmer, S., Torfah, H.: A stream-based specification language for network monitoring. In: Falcone, Y., Sánchez, C. (eds.) Runtime Verification. pp. 152–168. Springer International Publishing (2016)
14. Faymonville, P., Finkbeiner, B., Schledjewski, M., Schwenger, M., Stenger, M., Tentrup, L., Torfah, H.: Streamlab: Stream-based monitoring of cyber-physical systems. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification. pp. 421–431. Springer International Publishing, Cham (2019)
15. Faymonville, P., Finkbeiner, B., Schwenger, M., Torfah, H.: Real-time stream-based monitoring. ArXiv **abs/1711.03829** (2017)
16. Finkbeiner, B., Sipma, H.: Checking finite traces using alternating automata. Form. Methods Syst. Des. **24**(2), 101–127 (Mar 2004). https://doi.org/10.1023/B:FORM.0000017718.28096.48
17. Gorostiaga, F., Sánchez, C.: Striver: Stream runtime verification for real-time event-streams. In: Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings. pp. 282–298 (2018). https://doi.org/10.1007/978-3-030-03769-7_16
18. Havelund, K., Rosu, G.: Synthesizing monitors for safety properties. In: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 342–356. TACAS '02, Springer-Verlag, Berlin, Heidelberg (2002), http://dl.acm.org/citation.cfm?id=646486.694486
19. Hindy, H., Brosset, D., Bayne, E., Seeam, A., Tachtatzis, C., Atkinson, R.C., Bellekens, X.J.A.: A taxonomy and survey of intrusion detection system design techniques, network threats and datasets. ArXiv **abs/1806.03517** (2018)
20. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. **2**(4), 255–299 (Oct 1990). https://doi.org/10.1007/BF01995674
21. Lee, I., Kannan, S., Kim, M., Sokolsky, O., Viswanathan, M.: Runtime assurance based on formal specifications. In: In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (1999)

22. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: Tessla: Runtime verification of non-synchronized real-time streams. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing. pp. 1925–1933. SAC '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3167132.3167338
23. Li, J., Maier, D., Tufte, K., Papadimos, V., Tucker, P.A.: No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. SIGMOD Rec. **34**(1), 39–44 (Mar 2005). https://doi.org/10.1145/1058150.1058158
24. Liao, H.J., Lin, C.H.R., Lin, Y.C., Tung, K.Y.: Intrusion detection system: A comprehensive review. Journal of Network and Computer Applications **36**(1), 16 – 24 (2013). https://doi.org/https://doi.org/10.1016/j.jnca.2012.09.004
25. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Formal Modelling and Analysis of Timed Systems,FORMATS 2004, Grenoble, France, September 22-24, 2004, Proceedings. pp. 152–166 (2004). https://doi.org/10.1007/978-3-540-30206-3_12
26. Nickovic, D., Maler, O.: Amt: A property-based monitoring tool for analog systems. In: Raskin, J.F., Thiagarajan, P.S. (eds.) Formal Modeling and Analysis of Timed Systems. pp. 304–319. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
27. Paxson, V.: Bro: A system for detecting network intruders in real-time. In: Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7. pp. 3–3. SSYM'98, USENIX Association, Berkeley, CA, USA (1998), http://dl.acm.org/citation.cfm?id=1267549.1267552
28. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: A hard real-time runtime monitor. In: Proceedings of the 1st Intl. Conference on Runtime Verification. LNCS, Springer (November 2010)
29. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science. pp. 46–57. SFCS '77, IEEE Computer Society, Washington, DC, USA (1977). https://doi.org/10.1109/SFCS.1977.32
30. Reinbacher, T., Rozier, K.Y., Schumann, J.: Temporal-logic based runtime observer pairs for system health management of real-time systems. In: Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Grenoble, France, April 5-13. pp. 357–372 (2014). https://doi.org/10.1007/978-3-642-54862-8_24
31. Roesch, M.: Snort - lightweight intrusion detection for networks. In: Proceedings of the 13th USENIX Conference on System Administration. pp. 229–238. LISA '99, USENIX Association, Berkeley, CA, USA (1999), http://dl.acm.org/citation.cfm?id=1039834.1039864
32. Rozier, K.Y., Schumann, J.: R2U2: tool overview. In: RV-CuBES. Kalpa Publications in Computing, vol. 3, pp. 138–156. EasyChair (2017)
33. Sánchez, C.: Online and offline stream runtime verification of synchronous systems. In: Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings. pp. 138–163 (2018). https://doi.org/10.1007/978-3-030-03769-7_9
34. Schirmer, S., Benders, S.: Using runtime monitoring to enhance offline analysis. In: Proceedings of the Workshops of the Software Engineering Conference 2019, Stuttgart, Germany, February 19, 2019. pp. 83–86 (2019), http://ceur-ws.org/Vol-2308/aviose2019paper05.pdf
35. Torens, C., Adolf, F., Faymonville, P., Schirmer, S.: Towards intelligent system health management using runtime monitoring. In: AIAA Information Systems-AIAA Infotech @ Aerospace. American Institute of Aeronautics and Astronautics (AIAA) (jan 2017). https://doi.org/10.2514/6.2017-0419