Nested Monitors: Monitors as Expressions to Build Monitors

Felipe Gorostiaga[1,2,3(✉)] and César Sánchez[1]

[1] IMDEA Software Institute, Madrid, Spain
{felipe.gorostiaga,cesar.sanchez}@imdea.org
[2] Universidad Politécnica de Madrid (UPM), Madrid, Spain
[3] CIFASIS, Rosario, Argentina

Abstract. Stream runtime verification (SRV) is a formalism to express monitors as relations between typed input streams (observations) and typed output streams (data verdicts). In SRV, the actual data operations are separated from the temporal dependencies, therefore generalizing monitoring algorithms for temporal logics into the computation of richer verdicts. In this paper we study a new and powerful feature, which consists of lifting the execution of monitors to functions that can be used in defining expressions of enclosing specifications. At runtime, the outer monitor invokes the inner monitor passing a list of input events, called a *slice*. We present nested monitors for synchronous streams and for real-time event streams, allowing the elegant description of many specifications of interest, while still keeping the resources bounded.

We formally describe nested monitors and slices, and illustrate the practical application in many real-life examples, including electrocardiogram analysis (QRS), quantitative Metric Temporal Logic and arbitrary robustness of Signal Temporal Logic specifications.

1 Introduction

Runtime verification (RV) is a dynamic technique for software quality assurance that consists of generating a monitor from a formal specification. At runtime, the monitor inspects traces of the execution of the system under analysis, detecting violations of the specification. Motivated by the counterparts in static verification, early RV specification languages were based on temporal logics [3,14,28], regular expressions [35], timed regular expressions [1], rules [2], or rewriting [33]. Stream runtime verification (SRV), pioneered by Lola [10], defines monitors declaratively by equations that define the dependencies between output streams of results and input streams of observations, where the types of the streams and operations can be rich types of data. Unlike logical techniques, that compute Boolean verdicts, SRV allows rich observations and verdicts.

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

L. Feng and D. Fisman (Eds.): RV 2021, LNCS 12974, pp. 1–20, 2021.
https://doi.org/10.1007/978-3-030-88494-9_9

Examples include counting events, specifying and computing robustness values, generating models, quantitative verdicts and calculating target spatial coordinates. See [10,11,17,23] for examples illustrating the expressivity of SRV languages. The keystone of SRV is to separate two concerns: the temporal dependencies and the data manipulated. The temporal dependencies are inspired by the algorithms to monitor temporal logics which essentially capture the order of operations in monitoring algorithms. The data manipulation describes how to perform each individual operation and each element of storage that the monitor handles.

Different temporal algorithms exist for different notions of time. Early SRV works consider streams to be synchronized sequences of data (like in LTL semantics), so data observed in different streams at the same index in their sequences are considered to have occurred at the same time. Examples of synchronous SRV formalisms include the original Lola [10] and systems like Copilot [31]. Other formalisms that can be easily described using SRV include Mission-time LTL [32] and Functional Reactive Programming (FRP) [16]. Synchronous languages, like Lustre [27], Esterel [5] and Signal [21] also define relations between input and output values but these are designed to express behaviors so they assume causality and forbid future references, while in SRV future references are allowed to describe monitors that depend on future observations.

There have been approaches to extend SRV to real-time event streams, including RTLola [19], TeSSLa [9] and Striver [23], which consider streams to be sequences of timed events. Events contain data and are time-stamped with the instant of time at which the data is produced (either observed or generated). The time stamps of a stream must be monotonically increasing, but the events at a given index in two different streams can have arbitrary time stamps. These formalisms are known as asynchronous or real-time SRV. See [22] for a expressiveness comparison between synchronous time and asynchronous SRV. The data stored and computed is modeled as data types (data theories in the jargon of SRV) whose implementation is independent of the model of time. Most system include a handful of wired data types (e.g. [9,10,18]) but others study how to transparently incorporate data-types from programming languages [6].

The first contribution of this paper is *nested specifications*: using specifications to create a new data type which can be used in enclosing specifications. In this manner one can write new functions on sequences of data as an SRV specification which is invoked dynamically. The idea to decompose monitors into sub-monitors is not new. For example, in [12] the authors automatically derive cooperating monitors from a given definition, but this technique does not add expressivity to the data type language. The main concern in [12] is decentralized and distributed execution, which has also been studied in the context of RV [4,15,20] and SRV [11]. In this paper we consider expressivity and not decentralized execution. Nested specifications are particularly useful (1) when the caller monitor can invoke nested monitors with sub-traces of the original trace, which we model using *slices*; and (2) when the slice can be processed incrementally by the nested monitor to anticipate the computation of its ver-

dict. Nested specifications can be used both in synchronous and asynchronous SRV languages. Slices in languages like Lustre or Python allow dealing with collections of values in a convenient way, but these collections cannot refer to values of streams at different moments in time. Our slices are more similar to the notion of trace slicing in RV [8], in which slices correspond to sub-traces of a trace. We allow the definition and the early manipulation of slices that are partially known because some of their elements will only be known in the future. We show an implementation of these extensions for the formalisms Lola and Striver as reference languages of synchronous and asynchronous SRV respectively, (but the core ideas can be applied to other SRV formalisms easily). In synchronous SRV, the definition of a stream s can refer to future element of s or of other streams, with a syntactic restriction to ensure that every self-referential stream exclusively depends on previous or on future values. If all the self-referential streams in a specification depend exclusively on previous values, the specification is called *efficiently monitorable*. Specifications that only use present and past offsets, in which every stream is resolved immediately, are known as *very efficiently monitorable*. Note that efficiently monitorable specifications still allow future references as long as all self-referential streams end-up referring to past values. In efficiently monitorable specifications for every stream there is a constant upper-bound on time after which the stream will be resolved (the *latency* of the stream), a property known as *bounded lag* (as a particular case, very-efficiently monitorable specifications are 0 bounded lagged). Efficiently monitorable specifications are *trace-length independent* and can be monitored with bounded resources that can be calculated statically [10,34]. In all previous SRV efforts, if cyclic future offsets are allowed, specifications are assumed to not have bounded lag, and are only given semantics for finite traces. A second contribution of this paper is a notion of *dynamic bounded lag* in which streams can refer to other streams in the future with an offset that is not bounded a-priori but that is determined dynamically, which guarantees that the amount of resources necessary to monitor a specification is *slice-sizes* dependent. If an upper bound for the sizes of the slices can be calculated beforehand, then the specification is again trace-length independent. Since dynamic bounded specifications have semantics for future unbounded time domains, these specifications cannot be expressed in previous SRV formalisms. A third contribution of this paper is two implementations of nested monitors in HLola [24] and in HStriver [25]. We illustrate how these novel features enable specifications using several examples including QRS complex detection, quantitative semantics of the Metric Temporal Logic (MTL) and robustness of Signal Temporal Logic (STL) specifications.

The rest of the paper is structured as follows. Section 2 briefly revisits two SRV languages, Lola (for synchronous time) and Striver (for real-time asynchronous event streams). In Sect. 3 we present nested monitors and slices and in Sect. 4 we illustrate these features in action in several examples. Section 5 presents an empirical evaluation. Finally, Sect. 6 concludes.

2 Preliminaries

We briefly revisit SRV for synchronous and for real-time event streams.

Time and Streams. A *synchronous stream* is a sequence of length L of values from a data domain, where L may be a finite number or ω for infinite sequences. We refer to the value at the n-th position in a sequence z as $z(n)$. For example, the sequence $co2 = [350, 360, 289, 320, 330]$ contains samples of the level of CO2 in the air (measured in parts-per-million). In this sequence $co2(0) = 350$ and $co2(2) = 289$. A *real-time stream* is a succession of events (t, d) where d is a value from a value domain (as in synchronous streams) and t is a time-stamp. Time-stamps are elements of a *temporal domain* (for example $\mathbb{R}$, $\mathbb{Q}$ or $\mathbb{Z}$), a set whose elements are totally ordered. The interpretation of the time domain is to serve as a common clock to all the streams manipulated by a monitor (inputs and outputs). Time-stamps in every legal event stream are monotonically increasing. Given an element t in the temporal domain of an event stream r, we use $r(t)$ to refer to the value with time-stamp t in r. For example, the event-stream $tv\_status = \{(1.5, \mathit{off}), (4.0, \mathit{on}), (6.0, \mathit{off}), (7.5, \mathit{on}), (8.0, \mathit{off})\}$ indicates when a television is turned *on* or *off*. The event $(4.0, \mathit{on})$ in tv_status indicates that the TV is switched *on* at time 4.0. In this paper we use *stream* both for synchronous and event streams when it is clear from the context.

Data Theories. Streams are typed using multi-sorted first-order theories (which we call data theories). A type has a collection of symbols used to construct expressions, together with an interpretation of these symbols. The domain of a type is the set of values. Theories are interpreted in the sense that every function symbol f is both a constructor—used to build expressions (in the term algebra used to build expressions)—and an evaluator (that produces actual elements in the domain, which is the semantic interpretation of the function). We assume that every type D has a constructor `if` $\cdot$ `then` $\cdot$ `else` $\cdot$ that given a Boolean expression and two expressions of type D constructs a term of type D.

Lola is a synchronous SRV language, whose specifications declare the relation between output sequences (verdicts) and input sequences (observations). Similarly, Striver is a real-time SRV language, whose specifications describe the relation between output event-streams and input event-streams. We describe these formalisms separately. Due to space constraints we only introduce these languages concisely. See [10,23,26] for rigorous formal descriptions of Lola and Striver.

Lola: SRV for Synchronous Streams. Given a set of (typed) stream variables, Lola *stream expressions* consists of:

(1) *offsets* $v[k, d]$ where v is a stream variable of type D, k is an integer number and d a value from D, and
(2) *function applications* $f(t_1, \ldots, t_n)$ using constructors f applied to previously defined stream expression $t_1, \ldots, t_n$ of the right types.

Note that constants are 0-ary functions symbols. A stream variable v represents a sequence of the domain of the type of v. The intended meaning of an offset

expression $v[-1, d]$ is to capture the value of sequence v in the previous position (or value d if there is no such previous position, that is, at the beginning). The particular case for an offset with $k = 0$ requires no default value as the index is always guaranteed to be within the range of the sequence, in which case we use $v[now]$. A Lola specification consists of a set I of input stream variables, a set O of output stream variables, and a set of defining equations, $y_i = e_i$, one per output variable $y_i \in O$ where e_i is a stream expression of the same type as y_i that can use stream variables from I and O. The defining equations describe the relation between input and output streams. The *dependency graph* of a specification captures the dependency between streams and is built as follows: (i) there is a vertex for every variable in $I \cup O$, and (ii) there is an edge from u to v of weight k if the expression $v[k, d]$ appears in the definition of u. Cycles of negative weight represent self-references to previous values, and cycles of positive weight represent self-references to future values. Legal specifications do not contain cycles of zero weight.

Example 1. The specification "*the mean level of CO2 in the air in the last 3 instants*" can be expressed as follows where **denom** calculates the number of instants that are taken into account:

```
input Double co2
output Double denom = (min 2 denom[-1|0]) + 1
output Double mean  = (co2[-2|0] + co2[-1|0] + co2[now]) / denom[now]
```

In this specification $I = \{\texttt{co2}\}$ and $O = \{\texttt{denom}, \texttt{mean}\}$. □

The semantics of Lola is defined in terms of valuations which consist of one sequence ρ_x for each stream variable x, all of the same length L (where L can be finite or ω). Note that $\rho_x(n)$ is the value at position n of sequence ρ_x. A valuation induces a unique evaluation of all expressions $[\![e_x]\!]$ as follows:

– For offsets: $$[\![v[i, c]]\!](j) = \begin{cases} [\![v]\!](j + i) & \text{if } 0 \leq j + i < L \\ c & \text{otherwise} \end{cases}$$

– For functions: $$[\![f(e_1, \ldots, e_k)]\!](j) = f([\![e_1]\!](j), \ldots, [\![e_k]\!](j))$$

Note that f on the left hand side of the semantic definition of a function represents the syntactic representation of the function while the f on the right hand side represents the function evaluator.

We say that a valuation ρ satisfies a Lola specification φ whenever every output variable y_x satisfies its defining equation e_x, i.e. when $y_x = [\![e_x]\!]$. See [10] for a more rigorous explanation of valuations, which cannot be included here due to space constraints. Note how these semantics capture when a candidate valuation is an evaluation model, but the intention of a Lola specification is to compute the unique output sequences given input sequences. Efficient online monitoring algorithms can be generated for a specification depending on its dependency graph. If there are only negative cycles the specification can be monitored efficiently with bounded resources, and every stream will be determined in bounded time (see [10,34]), a property called bounded lag. These specifications, called *efficiently monitorable* have semantics for finite and infinite streams. If there are

positive cycles, then the specification only has semantics for finite traces and resources cannot be bound statically. In this paper we give semantics for infinite traces, for a sub-class of specifications with dynamic bounded future accesses, which we call *dynamic bounded lag*.

Striver: SRV for Real-Time Event Streams. Striver adapts Lola to real-time event streams. One step offsets are modified to fetch the next or previous event in the referred stream, and can be composed to access the next of the next event, the previous of the previous event, etc. Striver specifications define, for every output stream, (1) a *ticking* expression which captures when the stream may contain an event, as well as (2) a value expression that (just like in Lola) provides a value at the ticking instants. Concretely, there are three kinds of expressions in Striver:

- *Ticking Expressions*, which indicate the times at which a stream may produce an event. Ticking expressions can be $\{c\}$ for a specific time point c, or the ticking times of another stream s, unmodified with the operator **ticksOf** s, delayed by a constant k with **shift** k s, or delayed by the value of the stream itself with **delay** s. Ticking expression can be combined using $\cup$.
- *Offset Expressions*, which are the language construct that allows referring to the time instants when a given stream x contains an event. The basic offset expression is **t**, which represents the current instant. Given an offset expression τ and a stream x, we can build new offset expressions as follows. $x\mathbf{\ll}\tau$ represents the last instant at which x ticked in the past of τ. For example, the expression $x\mathbf{\ll}x\mathbf{\ll t}$ represents the second to last event of x with respect to the current time. The expression $x\mathbf{<\tilde{}}\tau$ is similar but also considers τ as a candidate instant. Analogously, $x\mathbf{\gg}\tau$ refers to the first future instant at which x contains an event (with $x\mathbf{>\tilde{}}\tau$ the variant that considers the present).
- *Value Expressions*, like in Lola, compute values. The simplest value expressions are constants to represent values in the domain of the type. Then, $x(\tau)$ provides the value of stream x at instant τ, which must be an offset expression for x. The atomic constructor **notick** is used to refrain from generating an event at a candidate ticking instant. The atomic expression **cv** allows accessing the value of an event in the ticking expression of the stream, useful when the event is shifted. Finally, values can be combined with constructor functions f of the appropriate types to compute new values.

We use x`[<t|d]` and x`[~t|d]` as syntactic sugar to refer to the most recent event in x (mimicking $x[-1, d]$ and $x[now]$ from Lola).

A Striver specification consists of one value expression V_y and one ticking expression T_y for each output stream of the appropriate type. For example, the property "*count for how long has the tv been on*" can be expressed as follows, where stream variable `tv_on` computes the result.

```
input TV_Status tv

output Int tv_on:
  ticks = ticksOf tv
  val = if tv[<t|off] == on then tv_on[<t|0] + (t - tv«t) else 0
```

The stream `tv_on` is computed at the times at which there is an event in `tv` as follows: its value is either

- 0 if the tv was previously off, or
- the previous value of `tv_on` plus `(t-tv[t)` (the difference in time from the previous time-stamp of a tv event) if the tv was previously on.

Therefore, the stream is tolerant to events that do not change the tv status. □

The semantics of Striver are also defined denotationally. Given real-time event streams for all events, ticking expressions a can be resolved to sets of instants $[\![a]\!]$, offset expressions can be resolved to instants in time (or the corresponding `-out` or `+out` value) and expressions e can be resolved to a data value $[\![e]\!]$, using the events fetched via offset expressions and the interpreted functions. Finally, a set of event-streams is an evaluation model if for every output stream variable y, the candidate event-stream ρ_y satisfies both the ticking equation and the value equation (see [23] for details):

$$\rho_y = \{(t, d) \mid t \in [\![\mathcal{T}_y]\!] \text{ and } d = [\![V_y]\!](t)\}$$

Similar definitions of dependency graph, efficient monitorability, etc. can be adapted for Striver specifications. Efficiently (trace length independent) online monitoring algorithms also exist for Striver for its efficiently monitorable fragment [23].

3 Nested Monitors and Slices

3.1 Nested Monitors and Slices in Lola

Slices. To introduce slices in Lola, we extend the syntax with the operator $x[:n]$, where x is a stream and n is an integer expression (which is not necessarily a constant). The semantics of the expression $x[:n]$ is the following:

$$[\![x[:n]]\!](j) \stackrel{\text{def}}{=} [\rho_x(j), \ldots, \rho_x(j+k)] \text{ where } k = [\![n]\!](j)$$

assuming that $k \geq 0$, otherwise the slice is the empty list []. Essentially, a slice is a consecutive sequence of n elements of x starting at the current position (where n is an integer value, for example, read from an integer input stream). Since a slice expression $x[:n]$ only refers to present and future values of x, this expression generates a dependency with a non-negative weight, which is not possible to calculate or bound statically. Therefore, we extend the dependency graph with a new kind of edge $y \rightarrow^+ x$ when the defining expression of y contains $x[:n]$. A $\rightarrow^+$ edge precludes the calculation of the *latency* of y and a bound to the memory required by the monitor. The notion of well-formedness in [10] states that "*a Lola specification is well-formed if there is no closed-walk with total weight zero in its dependency graph.*" Well-formed specifications are well-defined in the sense that for every collection of input sequences there is a unique collection of output sequences, which is a soundness requirement. For infinite

sequences, a specification is well-formed if every closed-walk in the dependency graph has a negative total-weight. Otherwise, it cannot be guaranteed that every expression has a unique value. Therefore, translating slice edges into a self-loop would disallow semantics for infinite traces. We adapt these definitions for slices.

Definition 1. *A Lola specification with slices is well-formed if it contains no zero-weight cycles and the sum of weights in any cycle containing slice edges is strictly positive. A well-formed Lola specification with slices is very-well-formed if no cycle contains a slice edge.*

The following lemma justifies this definition.

Lemma 1. *Let φ be a Lola specification with slices. If φ is well-formed then it is well defined for finite sequences. If φ is very-well-formed it is well defined both for finite and infinite sequences.*

The main idea of the proof is that in well-formed specifications for finite streams, and in very-well-formed specifications for infinite streams, a stream at a given instant only depends on a finite number of streams and positions.

Nested Monitors. Nested monitors allow spawning and executing monitors dynamically, collecting the result in each invocation and using it as a value in the caller monitor. This extension involves minimum changes to the language, because it mainly consists of lifting specifications to become new constructor symbols that extend data theories.

Consider the following specification, which calculates whether input numeric streams r and s will cross within the following 50 instants. We define a topmost specification as follows:

```
input Double r,s
output Bool willCross = runSpec (crossspec r[:50] s[:50])
```

The output stream `willCross` invokes the nested specification `crossspec` with the slices containing the next 50 events of `r` and `s` as input. We will usually use slices as input streams for inner specifications.

Defining an *inner* specification involves giving it a name and adding an extra clause: **return** x **when** y where x is a stream of any type and y is a Boolean stream. The type of the stream x determines the type of the value returned when the specification is invoked dynamically. Optionally, parameters can be provided when defining the inner specification. Once we have defined a specification *spec*, we can execute it using the reserved keyword *runSpec*, providing the necessary parameters and lists of values for the input streams, in the order in which they are defined in the inner specification. In our example, we define the nested specification `crossspec` as follows:

```
innerspec Bool crossspec

input Double r,s
output Bool cross = sgn(r[now] - s[now]) != sgn(r[-1|r[now]] - s[-1|s[now]])

return cross when cross
```

The output stream **cross** simply checks that the relative order of the streams **r** and **s** changes. When an inner specification with a return clause **return** x **when** y is executed, the computation will return the value of the stream x at the first time y becomes *true*, or the last value of x if y never holds in the execution. *As a consequence, if y becomes true in the middle of an execution, the monitor does not have to run until the end to compute a value and can anticipate the result.* This opens the door to evaluate the inner specification incrementally as new elements of the input slice are available, and return the outcome as soon as it is definite. We call this behavior *slice anticipation*. The maximum length of the input slice considered is determined by the minimum length of the inputs to the inner spec, which have to be finite and non-empty (as slices are), but they can have different lengths in different invocations.

The return clause in our example returns *true* as soon as a signal crossover is detected, and returns *false* if the stream **cross** never becomes *true*.

Example 2. The following specification reimplements the previous example using the type richness of the language to compute how far in the future the signals will crossover, avoiding running the inner specification for the following n instants:

```
input Double r,s
output Bool willCross = will[now] > 0
output Int will = if will[-1|0] > 1 then will[-1|] - 1 else runSpec crossspec2 r[:50] s[:50]
```

In this case, we define the output stream **willCross** as equivalent to the fact that the streams **will** cross in the future. The definition of the intermediate stream **will** works in two stages. First, if at the previous instant we knew there was going to be a crossover in $n > 1$ instants, then it returns $n - 1$. Otherwise, it invokes the inner monitor with the slices to calculate how far in the future the streams will cross (and if they will cross at all).

```
innerspec Int crossspec2

input Double r,s
output Bool cross = sgn(r[now] - s[now]) != sgn(r[-1|r[now]] - s[-1|s[now]])
output Int instantN = instantN[-1|0] + 1
output Int ret = if cross[now] then instantN[now] else 0

return ret when cross
```

The inner specification returns the current instant (on its own recollection of time) at which a crossover happens, and 0 if there is no crossover. The returned instant number at which there will be a crossover within the inner specification indicates in the outer specification how far in the future it will happen. □

Impact of the Extensions. In Lola [10,34] if the dependency graph of a specification contains cycles of positive weight, then the specification is *non-efficiently monitorable*, which means that some output streams may require waiting an unbounded number of instants to be resolved. The incorporation of slices introduces a new class of specifications: those in which the resources necessary for the computation of every value in the output streams can be calculated at runtime when the value is about to be computed. We call these *dynamic bounded lag* specifications. Note that efficiently monitorable specifications and very efficiently monitorable specifications are dynamic bounded lag (the value is even

known statically). *Past-time specifications (which contain no cycles of positive weight) that use slices are dynamic bounded lag specifications* but are not necessarily efficiently monitorable.

We say that specifications that are not dynamic bounded lag are *unbounded resource* specifications. The following table summarizes the new classification of specifications, where we mark the new class identified:

Dependency graph	**Class**
Only negative edges	**Very efficiently monitorable**
Only negative cycles, no $\cdot \rightarrow^{+} \cdot$ edges	**Efficiently monitorable**
Only negative cycles	**Dynamic bounded lag**
Any legal graph	**Unbounded resources**

3.2 Extensions in Striver

Using the rich expressive power of Striver we can define a stream `ws` that contains the events of a stream `s` in a window of length `w` as shown in the following program on the left. The output stream `ws` updates the list of events when an event of `s` is leaving the sliding window of events (i.e., when `s` is producing a value); and also uses the `shift` operator to retrieve the future values of the stream `s` and incorporate them to the sliding window. As a consequence, it is not necessary to extend Striver with an extra operator to implement slices as slices can be implemented by a simple translation defining a parametric auxiliary stream *slice* (a, b) x that returns the timestamped values of the stream x within the interval $(a, b]$ along with the last value of x before a. However, in practice *the incorporation of nested specifications and slices as libraries in the language greatly simplifies some stream definitions* when we let $x[a{:}b]$ be syntactic sugar to refer to slices.

```
output [(Time, a)] ws =
  ticks = ticksOf s U shift (-w) s
  val = let (mold, mnew) = cv
      prevls = ws [<t|[]]
      nextls = if mold == null then prevls
               else tail prevls
    in if mnew == null then nextls
       else nextls ++ [(t+w, mnew)]
```

The syntax for nested monitors in Striver is very similar to Lola. We define *inner* specifications with a name and an extra **return** clause, with the difference that the returned value may be null, if the returned stream x did not generate a value before the termination stream became *true*. Again, we run the defined inner specification using the function *runSpec*, providing the necessary parameters and lists of timestamped values for the input streams and, as in the case of Lola, we will usually use slices as input streams for inner specifications.

Example 3. Consider input numeric streams x and y, and the specification of whether the maximum value of x is lower than the minimum value of y within the following 50 time units. In the output stream `separable` we define two auxiliary streams `xs` and `ys` using slices and we use their values as the input streams of the inner specification `maxx_lt_miny`,

```
input Double x,y
output Bool separable:
  ticks = ticksOf xs U ticksOf ys
  val = runSpec
        maxx_lt_miny xs[~t|[]] ys[~t|[]]
  where xs = x[0:50] ; ys = y[0:50]
```

which is defined as follows. In the inner specification we define one auxiliary stream `max_x` to calculate the historical maximum of `x` (within the slice) and one auxiliary stream `min_y` to calculate the historical minimum of `y`.

```
innerspec Bool maxx_lt_miny
input Double x,y
output Double max_x:
  ticks = ticksOf x
  val = max cv max_x[<t|cv]
output Double min_y:
  ticks = ticksOf y
  val = min cv min_y[<t|cv]

output Bool ret:
  ticks = ticksOf min_y U ticksOf max_x
  val = max_x[~t|infty] < min_y[~t| infty]
output Bool stop:
  ticks = ticksOf ret
  val = not cv

return ret when stop
```

The output Boolean stream `ret` checks that the historical maximum of `x` stays below the minimum of the historical minimum of `y`. The specification returns *false* as soon as the property is violated, and *true* if it always holds. □

4 Nested Monitors and Slices in Action

We have extended two implementations of Lola and Striver (namely HLola and HStriver[1]) with nested monitors and slices. Both HLola and HStriver were written in Haskell and compiled with the version Glasgow Haskell Compiler, version 8.6.5 (see [6,24,25] for implementation details). In this section we illustrate how nested monitors and slices work in practice. A more advanced use case is the implementation of Kalman filter to predict trajectories of UAV in actual flight missions, described in [37].

QRS Complex Detection. We have implemented an electrocardiagram (ECG) analysis algorithm, in particular the Pan–Tompkins algorithm for real time QRS complex detection [30] (following [36] as reference ad-hoc code). Following our specification we can monitor the signal online, with an amount of memory that can be calculated statically. The specification declares output streams that iteratively transform the input signal as determined by the algorithm using statistics to detect peaks. We show below a snippet of the specification. The full specification[2] is 44 lines (while the reference implementation in Python uses 316 lines).

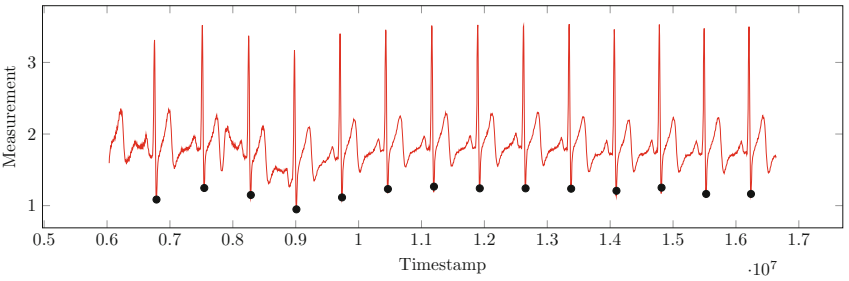
```
use innerspec headismax

input Double ecg_measurement
input Int timestamp

define Double convolved = ...
define [Double] rprev50 = shift rprev50[-1| replicate 50 (-0.000001)] convolved[now]
  where shift r x = x:init r
define Bool peak_candidate = convolved[now] > 0.35
                           && headisspike rprev50[now] && headisspike convolved[:50]
  where headisspike slice = runSpec (headismax slice)
define Bool ispeak = ...
define Bool isqrs  = ...
output Bool is_qrs_peak = isqrs[now] && ispeak[now]
```

[1] HLola and HStriver are available at http://github.com/imdea-software.

[2] Available at https://software.imdea.org/hlola/specs.html along with examples input and output events.

The specification follows an architecture of pipes and (stateful) filters. The stream **convolved** performs a convolution over a sliding window of fifteen instants of the square of the successive input value differences. A nested monitor checks if the current value of **convolved** is greater than all its previous 50 values (stored in **rprev50**) and its following 50 values, passed in **peak_candidate**. Finally, the output stream **is_qrs_peak** indicates if there is a peak in the ECG. The inner specification that assesses if the first value of an input stream is the maximum of the whole trace is shown on the left. Due to slice anticipation, the monitor produces the values at every instant as soon as possible. The following plot shows an ECG input signal and the peaks that the monitor detects (as black dots).

```
innerspec Bool headismax
input Double vals
output Double head = head[-1|vals[now]]
output Bool   ret  = head[now] >= vals[now]
output Bool   stop = not ret[now]
return ret when stop
```



MTL and Dynamic MTL. It is well known that Lola can define MTL properties by unrolling intervals. However, the size of the resulting specification is proportional to the width of the intervals. Using the nested monitors and slices, we have written an MTL library that allows specifications of constant size regardless of interval width. With these extensions, the width of the window can be of variable lookahead and use adaptive memory, as we show in Sect. 5. Moreover, the result at every instant is computed as soon as the minimum information is available, many times without requiring the whole interval (responsiveness speed is preserved).

```
library DynMTL
use innerspec mtluntilspec
output Bool until <(Int, Expr Int) (a,eb)> <Stream Bool phi> <Stream Bool psi> =
  untilaux (a,eb) phi psi [a| until (0, eb + a) phi psi[now]]
output Bool untilaux <(Int, Expr Int) (a,eb)> <Stream Bool phi> <Stream Bool psi> =
  let winphis = phi [:eb-a] ; winpsis = psi [:eb-a]
  in runSpec (mtluntilspec winphis winpsis)
```

The parametric stream **untilaux** creates two slices of the future values of φ and ψ and runs the nested monitor **mtluntilspec** over them. The parametric stream **until** shifts the values of **untilaux** and handles the corner cases. The inner specification **mtluntilspec** is defined on the left. This specification simply returns the value of ψ whenever it becomes *true* or φ becomes *false*. If this never happens, or if it happens at the last instant, the last value of ψ is returned.

```
innerspec Bool mtluntilspec
input Bool phi, psi
output Bool stop =     psi[now] ||
                   not phi[now]
return psi when stop
```

QMTL. We have defined a library for Quantitative MTL (QMTL), a more general version of MTL, which encompasses the original qualitative semantics, and provides quantitative semantics as well [7]. We first have defined a class in Haskell, shown on the left, to encapsulate the concepts of a Lattice and we have given two instances of lattices for Boolean and quantitative semantics. A Lattice consists of two operators: ⊔ and ⊓; with optional absorbing elements ⊤ and ⊥ respectively. We then define an instance for `Bool` and an instance for every numeric type as follows:

```
class Lattice x where
  sqcup     :: x -> x -> x
  sqcap     :: x -> x -> x
  opt_top   :: Maybe x
  opt_bottom :: Maybe x
```

```
instance Lattice Bool where
  sqcup     = (||)
  sqcap     = (&&)
  opt_top   = Just True
  opt_bottom = Just False
```

```
instance (Ord a, Num a) => Lattice a where
  sqcup     = max
  sqcap     = min
  opt_top   = Nothing
  opt_bottom = Nothing
```

We finally define the library for QMTL for any Lattice:

```
library QMTL

use innerspec foldspec
use innerspec foldaccumspec

output Lattice a => a eventually <(Int, Int) (x,y)> <Stream a phi> = let
  win = slidingwin (x,y) phi in
  if win == [] then opt_bottom else runSpec (foldspec sqcup opt_top win)
output Lattice a => a always <(Int, Int) (x,y)> <Stream a phi> = let
  win = slidingwin (x,y) phi in
  if win == [] then opt_top else runSpec (foldspec sqcap opt_bottom win)
output Lattice a => a since <(Int, Int) (x,y)> <Stream a phi> <Stream a psi> = let
  phis = slidingwin (x,y) phi
  psis = slidingwin (x,y) psi
  in runSpec (foldaccumspec sqcup sqcap opt_top phis psis)
output Lattice a => a since_overline <(Int, Int) (x,y)> <Stream a phi> <Stream a psi> = let
  phis = slidingwin (x,y) phi
  psis = slidingwin (x,y) psi
  in runSpec (foldaccumspec sqcap sqcup opt_bottom phis psis)
```

In this specification, `eventually` and `always` aggregate the values in the sliding window with the operators ⊔ and ⊓ respectively. If the absorbent element of the corresponding operator is found in the middle of the slice, then it is returned immediately as an optimization. This behavior is captured by the nested specification `foldspec`:

```
innerspec a foldspec <(a->a->a) op> <Maybe a mabs>

input a vals

output a ret = if instantN[now] === 1 then vals[now] else op ret [-1|] vals[now]
output Bool stop = ret[now] === abs

return ret when stop
```

Similarly, `since` and `since_overline` maintain the consecutive operation of φ along the window and combine it the current value of ψ, aggregating the results. If the value to return becomes the absorbent element mid-trace, then it is returned immediately, captured by the nested specification

```
innerspec a foldaccumspec <(a->a->a) op1> <(a->a->a) op2> <Maybe a mabs>
input a phi
input a psi
output a accum_phi = if instantN[now] === 1 then phi[now] else op2 accum_phi[-1|] phi[now]
output a ret = let val = op2 psi[now] accum_phi[now] in
   if instantN[now] === 1 then val else op1 ret[-1|] val
output Bool stop = ret[now] === abs
return ret when stop
```

The concrete types are instantiated automatically for each use of the library.

Robustl STL. We use the extensions in HStriver to define the quantitative semantics of the Signal Temporal Logic STL [13]. We show here $x\,\mathcal{U}_{[a,b]}\,y$ for which we define two slices: the slice of the events of x in $[t, t+b]$ and the slice of the events of y in $[t+a, t+b]$. Whenever an event enters or leaves any of the slices, we need to recompute the value of the stream. Since we treat the events of a stream as its change points, we use the value of the last event in the past of the slices to remember the value of the signal at the beginning of the sliding window, which we add to the slice timestamping it with t in x and $t+a$ in y. Finally, we execute the nested specification `robustuntilspec` with the resulting slices. This specification maintains the historical minimum of x (within the slice), compares them with each value of y, and returns the historical maximum of the results.

```
innerspec Int robustuntilspec
input Int xs, ys
output Bool never:
  ticks = {0}
  val = False
output Int xmins:
  ticks = ticksOf xs
  val = min xmins[<t|maxBound] cv
output Int theMins:
  ticks = ticksOf xmins U ticksOf ys
  val = min ys[~t|maxBound] xmins[~t|maxBound]
output Int theMaxMin:
  ticks = ticksOf theMins
  val = max theMaxMin[<t|minBound] cv
return theMaxMin when never
```

```
library RobustSTL
use innerspec robustuntilspec
define Int until <(Time, Time) (a,b)>
                 <Stream Int x>
                 <Stream Int y>:
  ticks = ticksOf xs U ticksOf ys
  val = runSpec (robustuntilspec xls yls)
  where
    xs = x[0:b]
    ys = y[a:b]
    xls = stampFst t xs[~t|(Nothing, [])]
    yls = stampFst (t+a) ys[~t|(Nothing, [])]
    stampFst _ (Nothing, r) = r
    stampFst ts (Just v, r) = (ts,v):r
```

5 Empirical Evaluation

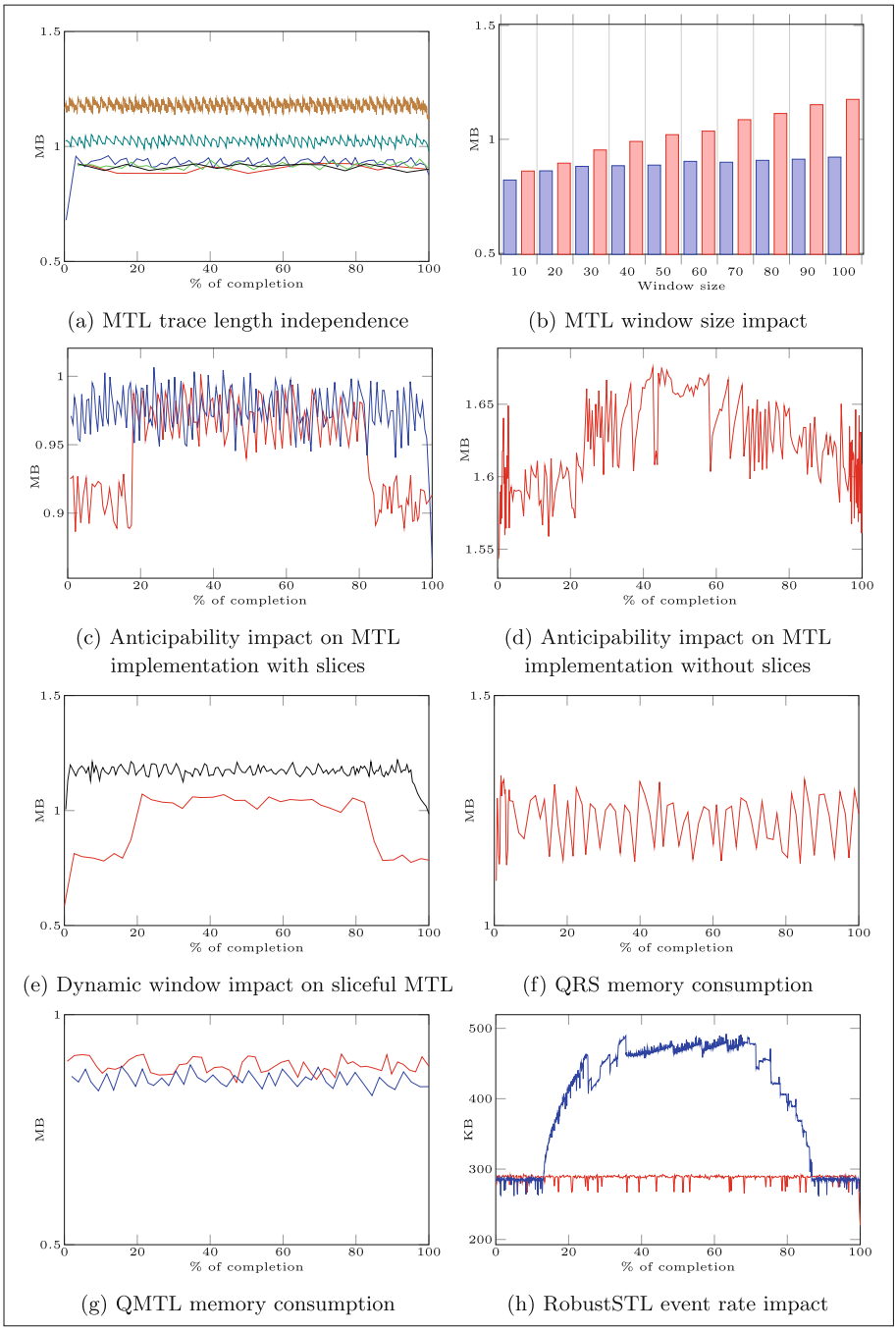
In this section we report an empirical evaluation, executed on a MacBook Pro with a Dual Core Intel-i5 at 2.5 GHz with 8 GB of RAM running MacOS Catalina. We evaluate empirically the following hypotheses:

- (H1) An MTL specification uses constant memory throughout its execution if the data does not allow slice anticipation in the middle of an interval. This holds for both the original and the new version of the MTL library.
- (H2) The direct implementation of MTL that does not use slices consumes more memory than the implementation of MTL with slices.

- (H3) In both versions of MTL the memory consumption is affected by whether the result can be slice-anticipated. If (H2) holds, this is even more prominent in the sliceful version since memory consumption is smaller. Therefore, memory consumption is only upper-bounded by a constant and the actual (lower) memory usage cannot be fully predicted.
- (H4) For the sliceful version of MTL, a dynamic variation in the size of a slice for a trace of non-slice-anticipable data has an impact on the memory consumption.
- (H5) The memory consumption for ECG oscillates periodically within a range due to slice anticipation, as a special case of (H3).
- (H6) The concrete type instantiation in the usage of the QMTL library does not affect memory consumption.
- (H7) Unlike in Lola, the memory consumption of running a RobustSTL specification in Striver, with fixed window size and non-slice-anticipable data, depends on the event rate. With a fixed event rate, the memory consumption is constant, while with a varying event rate, the memory consumption varies accordingly.

To evaluate these hypotheses we have carried out the following experiments.

- **Experiment MTL I.** In this experiment, we run the MTL specification $\varphi \, \mathcal{U}_{(0,w)} \, \psi$ with window sizes $w = 10, 50, 100$ with data that prevents slice anticipation and we measure the memory as input is processed. We replicate the experiments for both versions of the MTL library, the naive version with unrolling and the version with slices. The results are shown in Fig. 1(a) which shows that memory consumption is constant for any window size and library implementation, validating (H1). We also observe that memory consumption for $w = 50, 100$ using the original MTL implementation (the two topmost lines in the graph) is greater, which validates (H2).
- **Experiment MTL II.** In this experiment, we run the previous MTL specification with window sizes $w = 10, 20, \ldots, 100$ over data that prevents slice anticipation in both versions of the MTL library and we measured the average memory consumption. The results are shown in Fig. 1(b). The blue bars show the memory usage for the sliceful version of MTL, and the red bars, the memory usage for the direct version that does not use slices. The figure shows that memory grows with the window size, but it does so more rapidly for the MTL version with no slices, which validates (H2).
- **Experiment MTL III.** This experiment studies the impact of slice anticipation on memory usage. We run the previous MTL specification with a window size of $w = 250$ for both versions, with data that is immediately anticipable at the beginning and at the end of the trace, and data that is not anticipable in the middle. As can be seen in Fig. 1(c) and (d), the memory consumption grows when the value is not susceptible to slice anticipation, confirming (H3). For the sliceful version of the MTL library with only non-anticipable data, reported in Fig. 1(c), the memory consumption is bounded by the worst case scenario, represented by the blue line, which fluctuates in the range $[0.95, 1]$.



(a) MTL trace length independence

(b) MTL window size impact

(c) Anticipability impact on MTL implementation with slices

(d) Anticipability impact on MTL implementation without slices

(e) Dynamic window impact on sliceful MTL

(f) QRS memory consumption

(g) QMTL memory consumption

(h) RobustSTL event rate impact

Fig. 1. Outcomes of the empirical evaluation (Color figure online)

- **Experiment MTL IV.** In this experiment we run the monitor for the MTL specification with dynamically varying window (which can only be done for the slicefull version) and non-anticipable data. We set a window size w of 5 at the beginning and at the end of the trace, and 500 in the middle. This results in an increase of the memory consumption, as shown in Fig. 1(e). We have also included a run with a constant $w = 500$ to show that it is a boundary on the memory requirement. This experiment confirms (H4).
- **Experiment ECG.** This experiment consists on the analysis of the memory consumption of the QRS complex detection for ECG. The result is shown in Fig. 1(f), which shows that the memory increases and decreases periodically (within a certain range), due to the anticipability of the data every time a peak is found. This validates (H5).
- **Experiment QMTL.** This is the last experiment using HLola, where we run the QMTL specification $\varphi \, \mathcal{S}_{(0,10)} \, \psi$ for two instances of the `Lattice` class: `Bool` and `Double`, with non-anticipable data as input. The result is shown in Fig. 1(g), where we see that the concrete type does not affect the memory consumption, which confirms (H6).
- **Experiment HStriver (RobustSTL).** Finally, we have executed the RobustSTL specification $\varphi \, \mathcal{U}_{(0,5)} \, \psi$ using HStriver with varying event-rate in φ and ψ. The event rate is 2 events per seconds at the beginning and at the end of the trace, but 200 events per second in the middle, which results in an increase of the memory consumption, as shown in Fig. 1(h). We also show the execution of the specification with a constant event rate of 2, which results in a constant memory consumption. This confirms (H7).

6 Conclusions

We have introduced two extensions of SRV, nested specifications and slices, both for synchronous and real-time and implemented them in HLola and HStriver. These extensions make many specifications more concise and easier to read. In turn, we have captured a new class of *dynamically bounded lag* specifications, where streams can depend unboundedly on the future but still have semantics for infinite inputs. We have used these extensions to implement a QRS complex detection algorithm, MTL, QMTL and robustness specifications for STL. The empirical evaluation shows that memory usage is predictable and, in the case of MTL, outperforms previous implementations in those cases where previous implementations existed. Future work includes evaluating quantitative STL properties for a powertrain control verification from [29], where input signals are precomputed from a MatLab simulation.

References

1. Asarin, E., Caspi, P., Maler, O.: Timed regular expressions. J. ACM **49**(2), 172–206 (2002). https://doi.org/10.1145/506147.506151
2. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 44–57. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24622-0_5
3. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14 (2011). https://doi.org/10.1145/2000799.2000800
4. Bauer, A., Falcone, Y.: Decentralised LTL monitoring. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 85–100. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_10
5. Berry, G.: The foundations of Esterel. In: Proof, Language, and Interaction: Essays in Honour of Robin Milner, pp. 425–454. MIT Press (2000)
6. Ceresa, M., Gorostiaga, F., Sánchez, C.: Declarative stream runtime verification (hLola). In: Oliveira, B.C.S. (ed.) APLAS 2020. LNCS, vol. 12470, pp. 25–43. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64437-6_2
7. Chattopadhyay, A., Mamouras, K.: A verified online monitor for metric temporal logic with quantitative semantics. In: Deshmukh, J., Ničković, D. (eds.) RV 2020. LNCS, vol. 12399, pp. 383–403. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-60508-7_21
8. Chen, F., Roşu, G.: Parametric trace slicing and monitoring. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 246–261. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_23
9. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: TeSSLa: temporal stream-based specification language. In: Massoni, T., Mousavi, M.R. (eds.) SBMF 2018. LNCS, vol. 11254, pp. 144–162. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03044-5_10
10. D'Angelo, B.: LOLA: runtime monitoring of synchronous systems. In: Proceedings of the 12th International Symposium of Temporal Representation and Reasoning (TIME 2005), pp. 166–174. IEEE CS Press (2005). https://doi.org/10.1109/TIME.2005.26
11. Danielsson, L.M., Sánchez, C.: Decentralized stream runtime verification. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 185–201. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32079-9_11
12. Delaval, G., Girault, A., Pouzet, M.: A type system for the automatic distribution of higher-order synchronous dataflow programs. SIGPLAN Not. **43**(7), 101–110 (2008). https://doi.org/10.1145/1379023.1375672
13. Donzé, A., Ferrère, T., Maler, O.: Efficient robust monitoring for STL. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 264–279. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_19
14. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Van Campenhout, D.: Reasoning with temporal logic on truncated paths. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 27–39. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_3
15. El-Hokayem, A., Falcone, Y.: Monitoring decentralized specifications. In: Proceedings of the 26th ACM SIGSOFT Internaional Symposium on Software Testing and Analysis (ISSTA 2017), pp. 125–135. ACM (2017). https://doi.org/10.1145/3092703.3092723

16. Eliot, C., Hudak, P.: Functional reactive animation. In: Proceedings of ICFP 2007, pp. 163–173. ACM (1997). https://doi.org/10.1145/258948.258973
17. Faymonville, P., Finkbeiner, B., Schirmer, S., Torfah, H.: A stream-based specification language for network monitoring. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 152–168. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_10
18. Faymonville, P., et al.: StreamLAB: stream-based monitoring of cyber-physical systems. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 421–431. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_24
19. Faymonville, P., Finkbeiner, B., Schwenger, M., Torfah, H.: Real-time stream-based monitoring. CoRR abs/1711.03829 (2017)
20. Francalanza, A., Pérez, J.A., Sánchez, C.: Runtime verification for decentralised and distributed systems. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 176–210. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_6
21. Gautier, T., Le Guernic, P., Besnard, L.: SIGNAL: a declarative language for synchronous programming of real-time systems. In: Kahn, G. (ed.) FPCA 1987. LNCS, vol. 274, pp. 257–277. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-18317-5_15
22. Gorostiaga, F., Danielsson, L.M., Sánchez, C.: Unifying the time-event spectrum for stream runtime verification. In: Deshmukh, J., Ničković, D. (eds.) RV 2020. LNCS, vol. 12399, pp. 462–481. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-60508-7_26
23. Gorostiaga, F., Sánchez, C.: Striver: stream runtime verification for real-time event-streams. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 282–298. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_16
24. Gorostiaga, F., Sánchez, C.: HLola: a very functional tool for extensible stream runtime verification. In: Groote, J.F., Larsen, K.G. (eds.) TACAS 2021. LNCS, vol. 12652, pp. 349–356. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72013-1_18
25. Gorostiaga, F., Sánchez, C.: HStriver: a very functional extensible tool for the runtime verification of real-time event streams. In: FM 2021 (2021, to appear)
26. Gorostiaga, F., Sánchez, C.: Stream runtime verification of real-time event streams with the Striver language. Int. J. Softw. Tools Technol. Transfer **23**, 157–183 (2021). https://doi.org/10.1007/s10009-021-00605-3
27. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language LUSTRE. Proc. IEEE **79**(9), 1305–1320 (1991). https://doi.org/10.1109/5.97300
28. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46002-0_24
29. Jin, X., Deshmukh, J.V., Kapinski, J., Ueda, K., Butts, K.: Powertrain control verification benchmark. In: Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control (HSCC 2014), pp. 253–262. ACM (2014). https://doi.org/10.1145/2562059.2562140
30. Pan, J., Tompkins, W.J.: A real-time QRS detection algorithm. IEEE Trans. Biomed. Eng. **BME-32**(3), 230–236 (1985). https://doi.org/10.1109/TBME.1985.325532
31. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: a hard real-time runtime monitor. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 345–359. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_26

32. Reinbacher, T., Rozier, K.Y., Schumann, J.: Temporal-logic based runtime observer pairs for system health management of real-time systems. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 357–372. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_24
33. Roşu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Autom. Softw. Eng. **12**(2), 151–197 (2005). https://doi.org/10.1007/s10515-005-6205-y
34. Sánchez, C.: Online and offline stream runtime verification of synchronous systems. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 138–163. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_9
35. Sen, K., Roşu, G.: Generating optimal monitors for extended regular expressions. In: Sokolsky, O., Viswanathan, M. (eds.) Electronic Notes in Theoretical Computer Science, vol. 89. Elsevier (2003)
36. Sznajder, M., Łukowska, M.: Python online and offline ECG QRS detector based on the Pan-Tomkins algorithm, July 2017. https://doi.org/10.5281/zenodo.826614
37. Zudaire, S., Gorostiaga, F., Sánchez, C., Schneider, G., Uchitel, S.: Assumption monitoring using runtime verification for UAV temporal task plan executions. In: Proceedings of IEEE International Conference on Robotics and Automation (ICRA 2021). IEEE (2021)