

Two Ways to See the Future: Combining Prediction and Future-Offset Accesses in RTLola

Jan Baumeister¹, Bernd Finkbeiner^{1,2}, Eduard Müller¹,
Frederik Scheerer^{(✉)1}, and Julia Tillman³

¹ CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
{jan.baumeister,finkbeiner,eduard.mueller,frederik.scheerer}@cispa.de

² Technical University of Munich, Germany

³ Saarland University, Saarbrücken, Germany

Abstract. RTLola is a stream-based specification language designed for asynchronous real-time systems. While many temporal specifications naturally refer to future behavior, RTLola currently offers no mechanism to express such future-dependent properties. In this paper, we extend RTLola with two complementary mechanisms to reason about the future. First, we introduce a *prediction operator* that extrapolates future stream values at arbitrary timestamps based on past observations. Second, we add a discrete *future offset operator*, which provides access to precise future values by delaying the evaluation of the dependent stream expressions. While the former enables immediate, but possibly imprecise predictions, the latter ensures exact values once the required information becomes available. We formalize both extensions in the RTLola semantics and evaluate their implementation on runtime and memory consumption.

Keywords: Stream-Based Monitoring · Cyber-Physical Systems

1 Introduction

Many monitoring tasks naturally require reasoning about future system behavior. Such requirements can be expressed naturally in future-time temporal logic: for instance, the requirement that each takeoff event must be eventually followed by the aircraft reaching a valid flight altitude can be written in LTL as

$$\Box(\text{takeoff} \Rightarrow \bigcirc(\neg \text{takeoff} \mathcal{U} \text{alt_reached})).$$

In monitoring, specifications that refer to the past are usually preferable to those that reason about the future, because past-time specifications can be evaluated in an online manner without lookahead. For the monitorable fragments of logics like LTL, it is possible to translate future-time specifications into equivalent past-time specifications at the cost of an exponential blow-up in the formula [22,1]. For stream-based specification languages like RTLola [6], the situation is more difficult. Unlike temporal logics, which evaluate a single verdict for an entire trace, RTLola produces a new value at every time point. A future-to-past translation

would shift the evaluation points of the stream expressions, and with it, the timestamp at which a violation is reported. However, in a real-time monitoring context, the precise time of a detected event is important. For example, we might want the monitor to report the timestamp of the takeoff that triggered the violation, not the timestamp at which the violation was confirmed.

RTLola currently offers no mechanism to express future-time specifications. In this paper, we introduce two complementary mechanisms for accessing future information. A *prediction operator* for getting immediate, but possibly imprecise predictions, and a *future-access operator*, which gets exact, but delayed information. The prediction operator enables forecasting of future stream values based on a bounded history of past observations. At each step, a prediction function is applied to a sliding window of recent values to estimate the stream’s value at a future offset. This prediction function can be instantiated with standard models such as linear regression or other statistical and machine learning techniques. This design enables flexible integration of domain-specific forecasting methods while preserving a uniform interface at the specification level. However, such predictions are inherently uncertain. In highly dynamic scenarios, the predicted values may deviate significantly from the actual future values.

An alternative approach, which was already present in the original LOLA [14] language, is to access future values directly via a future-offset operator. This operator refers to values that will occur at a later point in time, providing exact information rather than estimates. Naturally, this comes at the cost of a delayed evaluation: results can only be produced once the referenced future values have been observed. Consequently, specifications using future offsets introduce latency but yield precise results. This future access operator complements the prediction operator. The two approaches provide a trade-off between immediacy and accuracy, and can be combined to obtain early estimates and precise confirmations. In contrast to LOLA, supporting future-offset semantics in RTLola is more difficult due to its asynchronous stream model, where streams may evolve at different rates and lack a global synchronized clock. As a consequence, future references are not sound in general. We identify a well-defined subset of RTLola specifications for which future-offset accesses are supported, and strengthen the language’s well-formedness criteria to guarantee this property. We evaluate our approach on runtime and memory usage, and show that both remain manageable and, in many cases, comparable to standard RTLola monitoring. Finally, we show that depending on the input data, our implementation can maintain bounded memory even for specifications that are unbounded in principle.

The remainder of this paper is structured as follows. Section 2 introduces the necessary background on RTLola. Section 3 presents the prediction operator, followed by the future offset operator in Section 4. Section 5 then reports on the experimental evaluation.

1.1 Related Work

Past-time and future-time specifications represent two established approaches to runtime monitoring [19,3]. While past-time operators enable incremental evaluation, future-time operators allow more natural and succinct specifications [1],

at the cost of delayed [14] or inconclusive verdicts [3]. Our approach extends stream-based runtime monitoring with two mechanisms for reasoning about future behavior: prediction and exact future accesses.

Prediction of future system behavior has been studied extensively in runtime monitoring and cyber-physical systems. Existing approaches rely on probabilistic and machine learning techniques, such as Bayesian estimation [12], neural network-based predictors [10], and general machine learning methods [11]. These methods typically require training data or domain specific knowledge. In contrast, lightweight statistical techniques like polynomial regression [15] enable forecasting without extensive training. Our work follows this approach, integrating predictive reasoning directly into the specification language RTLola. Predictive semantics for runtime verification have also been studied in the context of anticipating future property satisfaction or violation based on partial observations and system models [21,28,23,20].

Besides prediction, several monitoring frameworks support exact future references. In monitoring temporal logics, future-time operators are commonly used to reason about future system behavior [3,2,26]. Stream-based languages provide richer verdicts. In this setting, future accesses appear mostly in monitoring languages for synchronous systems [14,17,18] or in offline monitoring settings [9,24], where the complete trace is available during evaluation. One example for a synchronous stream-based language is LOLA [14], the predecessor of RTLola, which supports future-offset accesses in a synchronous setting with a global clock. In contrast, asynchronous stream-based languages [6,13] allow streams to evolve independently at different rates. While the asynchronous monitoring framework TeSSLa [13,25] supports scheduling events at future timestamps via delay, it does not provide explicit future-offset accesses, as considered in this work. Until now, the RTLola [6] semantics are restricted to evaluation based only on present and past information. Our work extends future-offset accesses from the synchronous setting of LOLA to the asynchronous setting of RTLola.

2 Background

RTLola [6] is a stream-based monitoring language. Stream-based monitors operate over *streams*, i.e., infinite sequences of values. *Input streams* continuously receive values of the monitored system. *Output streams* are defined through stream equations and derive new values by filtering and aggregating the data. *Trigger*, special boolean-valued output streams, indicate a specification violation.

Consider the specification in Figure 1 as an example. The input stream `alt` provides the readings of a drone’s altitude sensor. First, this measurement is converted to altitude above ground level by subtracting the ground-level height. To mitigate sensor noise, the output stream `alt_smoothed` computes a sliding-window average over the previous two seconds, evaluated periodically every second. The stream `alt_reached` evaluates whether the smoothed altitude exceeds a threshold of 10.0, and `alt_sustained` checks whether this condition holds for two consecutive evaluations. To achieve this, the stream accesses the previous

```

input alt : Float64
output alt_above_ground @alt := alt - GROUND_LEVEL
output alt_smoothed @1s :=
  alt_above_ground.aggregate(over: 2s, using: avg).defaults(to: 0.0)
output alt_reached @1s := alt_smoothed > 10.0
output alt_sustained @1s :=
  alt_reached ^ alt_reached.offset(by: -1).defaults(to: false)
trigger alt_sustained "altitude sustained"

```

Fig. 1: An RTLOLA specification monitoring an altitude sensor.

value of `alt_reached` using `offset`. Since no previous value exists at the beginning of monitoring, a default value is provided for this case. The trigger emits a notification once the condition is satisfied.

2.1 Types

Every stream in RTLOLA has two types. The *value type* describes the kind of data contained in the stream, such as a boolean, integer, or floating-point number. The *pacing type* determines *when* new stream values are computed. RTLOLA distinguishes between two kinds of pacing types. *Event-based streams* are evaluated whenever specific inputs receive new values, whereas *periodic streams* are evaluated at a fixed frequency. Both value and pacing type are often inferred automatically, but can also be annotated explicitly. For example, the annotation `@1s` on the stream `alt_smoothed` indicates that it is evaluated once per second, while the `alt_above_ground` stream is annotated with the input stream `@alt`, s.t. it is evaluated for every new arriving altitude reading.

Further, pacing types can include dynamic filter conditions. Assuming an additional boolean input `alt_reliable`, we can adapt the `alt_above_ground` stream to only produce values when the current altitude reading is reliable:

```

output alt_above_ground
  eval @alt ^ @alt_reliable when alt_reliable with alt - GROUND_LEVEL

```

Only if the `when` condition is satisfied, stream values are produced by evaluating the `with` expression. Notice that the `:=` syntax used before is just syntactic sugar for streams with a `when` condition of `true`.

In RTLOLA, we differentiate between synchronous and asynchronous stream accesses. All accesses of stream values encountered so far are *synchronous*, meaning that the pacing types guarantee that, whenever a stream is accessed, the accessed stream has a value available at exactly that point in time. For instance, the `alt_sustained` stream can access the `alt_reached` stream synchronously because both streams are evaluated once per second. In contrast, to make stream accesses across different-paced streams, we must use an *asynchronous* access. As an example, consider the `alt_display` stream, which outputs the most recent value from the `alt_above_ground` stream every second:

```
| output alt_display @1s := alt_above_ground.hold().defaults(to: 0.0)
```

This stream makes use of an `hold` access, which accesses the *most recent value* of the target stream, regardless of when that value was produced. Because this value could potentially not exist, a default value is required for this case. Sliding windows are another example of asynchronous stream accesses, such as on the `alt_smoothed` stream from Figure 1. For a more detailed introduction to the RTLola specification language, we refer the reader to the RTLola Tutorial⁴ [6].

2.2 Semantics

The semantics of RTLola are defined with respect to an *evaluation model*. We consider discrete timesteps $\text{Time} = \{n \in \mathbb{N} \mid n \leq T_{max}\}$ and a domain of optional stream values $\mathbb{V}_\perp = \mathbb{V} \cup \{\perp\}$.

Definition 1 (Evaluation Model [6,8]). *Let φ be an RTLola specification with a set of input stream references $\text{ID}^\uparrow$ and output stream references $\text{ID}^\downarrow$. An evaluation model $\omega \in \mathbb{W}$ is defined as:*

$$\begin{aligned} \text{Stream} &:= \text{Time} \rightarrow \mathbb{V}_\perp \\ \text{StreamMap} &:= \text{ID}^\uparrow \uplus \text{ID}^\downarrow \rightarrow \text{Stream} \\ \text{TimeMap} &:= \text{Time} \rightarrow \mathbb{R}^+ \\ \mathbb{W} &:= \text{StreamMap} \times \text{TimeMap} \end{aligned}$$

An evaluation model consists of a stream map and a time map. The stream map assigns each stream reference a time-indexed sequence of values. As RTLola is asynchronous, values may be absent at certain timesteps, denoted by $\perp$. The time map assigns discrete timestamps to real-time timestamps.

Given an evaluation model $\omega = (\text{streams}, \text{times}) \in \mathbb{W}$, we use the following shorthand notation: $\omega(t) := \text{times}(t)$ to return the real-time timestamp for a time $t \in \text{Time}$, and $\omega(s) := \text{streams}(s)$ returns the stream associated with stream reference $s \in \text{ID}^\uparrow \uplus \text{ID}^\downarrow$.

Definition 2 (RTLola Semantics [7]). *Given an RTLola specification φ , we define the semantics as the set:*

$$\begin{aligned} \llbracket \varphi \rrbracket &= \{ \omega \in \mathbb{W} \mid \forall \text{sid} \in \text{ID}^\downarrow. \forall t \in \text{Time}. \varphi(\text{sid}) \Downarrow_\omega^t \omega(\text{sid})(t) \\ &\quad \wedge \forall t \in \text{Time}. \omega(t) < \omega(t+1) \} \end{aligned}$$

In the formula, $\varphi(\text{sid}) \Downarrow_\omega^t v$ denotes that the defining stream equation $\varphi(\text{sid})$ of stream sid , evaluated at time t relative to the evaluation model ω , yields the value v . This evaluation first checks whether the stream's pacing condition is satisfied at time t . If so, the result is the value obtained from evaluating the stream's equation, and otherwise the result is $\perp$. This definition ensures that each output stream calculates its value according to the stream's equations $\varphi(\text{sid})$

⁴ <https://rtlola.cispa.de/playground/tutorial>

at every timestep t . Furthermore, the real-time timestamps must be strictly monotonically increasing. We additionally require that specifications are well-defined:

Definition 3 (Well-Defined Specification [27,7]). *An RTLOLA specification φ is well-defined, if, for every inputs stream assignment $I : \text{Time} \rightarrow (\text{ID}^\uparrow \rightarrow \mathbb{V}_\perp)$ and time map $T : \text{Time} \rightarrow \mathbb{R}^+$, there exists exactly one evaluation model $\omega \in \llbracket \varphi \rrbracket$ such that for all $t \in \text{Time}$,*

$$\omega(t) = T(t) \quad \text{and} \quad \forall \text{sid} \in \text{ID}^\uparrow. \omega(\text{sid})(t) = I(t)(\text{sid}).$$

In other words, a well-defined specification admits exactly one evaluation model for every possible input stream assignment. Consequently, all output stream values are uniquely determined by the input values and their timing.

2.3 Static Analysis

RTLOLA guarantees well-definedness by implementing a static analysis based on a dependency graph, which captures temporal dependencies between streams.

Definition 4 (Dependency Graph [27]). *Let φ be an RTLOLA specification. The corresponding dependency graph of φ is a directed multi-graph $G_\varphi = \langle V, E \rangle$ with $V = \text{ID}^\uparrow \uplus \text{ID}^\downarrow$. The set of edges is represented by $E \subseteq V \times L_D \times V$, where*

$$L_D = \{\text{Filter}, \text{Eval}\} \times (\{\text{Sync}\} \cup \{\text{Hold}\} \cup (\{\text{Offset}\} \times \mathbb{Z}) \cup (\{\text{Aggr}\} \times \mathcal{F} \times \mathbb{R}))$$

is the corresponding dependency label. The first component is the behavioral argument, and the second component is the stream access kind. An edge $(s, (b, k), s')$ is in E iff the stream expression of s contains a stream access to s' of kind k .

Figure 2 depicts the dependency graph for the specification in Figure 1. Each stream access in the specification corresponds to an edge in the graph. Since all accesses occur in **with**-expressions, every edge is annotated with *Eval*. Accesses in **when**-conditions would carry a *Filter* annotation instead.

The static analysis uses the dependency graph and is conducted along three dimensions. The *well-formedness* is a static criterion ensuring a well-defined specification. The *evaluation order* represents a partial order on streams and ensures that streams are efficiently resolved within one evaluation cycle. Intuitively, this ensures that all dependencies of a stream are evaluated before the stream itself within each evaluation cycle. Finally, the *memory bound* of a stream gives us the number of values that must be retained for monitoring. This computation ensures that the monitor operates within finite memory constraints. The existing definitions [27] are insufficient for the support of our proposed operators and are extended throughout the rest of this paper.

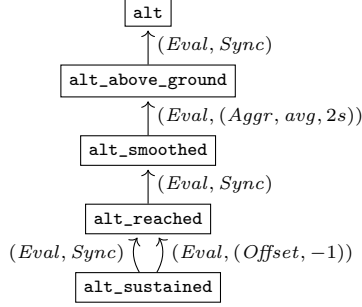


Fig. 2: The dependency graph for Figure 1.

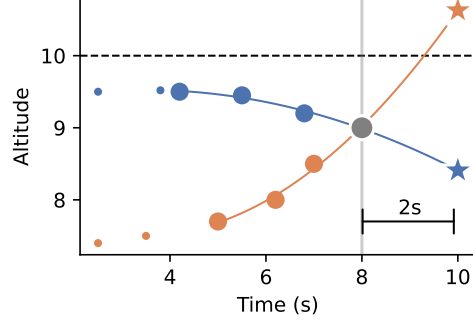


Fig. 3: An example of the prediction given two different altitude traces.

3 Prediction Operator

The predict operator enables the monitor to predict future stream values based on past observations. As an example, consider again the situation from Section 2 monitoring the altitude of a drone. By incorporating prediction, the monitor can issue early warnings when a violation is likely to occur in the near future, enabling countermeasures to be taken before the violation occurs. Figure 3 illustrates this behavior for two altitude traces (orange and blue). At each evaluation point, the prediction operator considers the last four observed values of the respective trace (highlighted in the figure) and fits a regression model to these points. This model is used to estimate the altitude value two seconds into the future. As shown, one trace yields a predicted violation while the other remains within safe bounds. This can be expressed directly in RTLola as follows:

```
output prediction @1s := alt_smoothed.predict(using: polyreg2,
                                              over_discrete: 4, in: 2s).defaults(to: 0.0)
trigger prediction > 10.0 "height violation expected soon"
```

Here, a second-degree polynomial is fitted to the four most recent altitude values, and the resulting model is evaluated at the specified future time point to obtain the predicted value. If not all four values exist yet, a default value is used instead.

Formally, we introduce prediction as a pure operator over finite sequences of timestamped values. The stream-based syntax presented above is then defined as syntactic sugar that extracts the required observations and applies the core operator. This separation decouples the prediction mechanism from RTLola's stream semantics and allows the existing static analyses to remain unchanged.

3.1 Core Prediction Operator

We first define a prediction function independently of RTLola's syntax. This definition is generic in the choice of a prediction model γ , requiring only that γ provides a function construction via $\gamma.fit$.

Definition 5 (Prediction Function). Let γ be a prediction model and $n \in \mathbb{N}_{>0}$. The prediction function γ_n constructs a function based on a sequence of n pairs of timestamps and values.

$$\gamma_n : (\mathbb{R}^+_{\perp} \times \mathbb{V}_{\perp})^n \rightarrow (\mathbb{R}^+ \rightarrow \mathbb{V}_{\perp})$$

$$\gamma_n((t_1, v_1), \dots, (t_n, v_n)) = \begin{cases} f_{\perp} & \text{if } \exists 1 \leq i \leq n. v_i = \perp \\ \gamma.\text{fit}((t_1, v_1), \dots, (t_n, v_n)) & \text{otherwise} \end{cases}$$

Here, $f_{\perp} : \mathbb{R}^+ \rightarrow \perp$ denotes the constant function returning $\perp$ for all timestamps. Otherwise, $\gamma.\text{fit}((t_1, v_1), \dots, (t_n, v_n))$ yields a function mapping real-time timestamps to predicted values.

Different instantiations of the prediction model γ correspond to different regression or extrapolation techniques. For instance, a linear regression model fits a line to the observed data points, while higher-order polynomial models capture nonlinear trends. In practice, the choice of model can be adapted to the monitoring application and the computational overhead that can be tolerated at runtime.

3.2 Syntactic Sugar

We integrate prediction into RTLOLA using a dedicated prediction expression.

Definition 6 (Prediction Operator). Let s be a stream, γ a prediction model, $n \in \mathbb{N}_{>0}$, and $\delta \in \mathbb{R}^+$ a positive real-time offset. The stream expression

`| s.predict(using:  $\gamma$ , over_discrete:  $n$ , in:  $\delta$ )`

evaluates at time t to $\gamma_n(v_{s,n,t})(\omega(t) + \delta)$, where $v_{s,n,t}$ denotes the sequence of the n most recent timestamped value pairs of s available at t .

Intuitively, the operator evaluates at each time t by extracting the n most recent timestamped values of s available at t , constructing a prediction function via γ_n , and evaluating this function at the future timestamp $\omega(t) + \delta$. If any of the selected points is $\perp$, the result is $\perp$ as well. For this case, the prediction must be followed by a default expression.

As syntactic sugar, we can make use of past `offset` expressions to extract the required points. For a stream s , we first collect the most recent n values together with their corresponding timestamps using discrete offset accesses:

```
output s_points_n :=
  (time(s).offset(by: -(n-1)), s.offset(by: -(n-1))),
  ...
  (time(s).offset(by: -1), s.offset(by: -1)),
  (time(s), s)
```

Here, `time(s)` denotes a stream that shares the same pacing as s but carries the corresponding timestamps of its evaluation points. Due to the synchronous


```

output takeoff @1s := alt_smoothed ≤ 0.05
  ∧ alt_smoothed.offset(by: 1).defaults(to: 0.0) > 0.05
output valid_flight := alt_reached ∨ (¬takeoff ∧
  valid_flight.offset(by: 1).defaults(to: false))
trigger ¬(takeoff ⇒ valid_flight.offset(by: 1).defaults(to: false))

```

Fig. 4: An RTLola specification monitoring consecutive flights of a drone for altitude thresholds.

nature of the offset access, the pacing of this derived stream inherits the pacing of s . Consequently, at every evaluation point of s , the tuple of the most recent n points is computed. Note that `offset` returns $\perp$ if the accessed value does not exist, which is then handled by the prediction function. A prediction expression as given in Definition 6 is translated into the core operator application

```

|  $\gamma_n(s\_points_n.\text{hold}())(\text{time} + \delta)$ 

```

where `time` gives access to the current timestamp of the stream evaluation. The `hold` access is required to decouple the pacing of s from the evaluation of the prediction expression. This allows the prediction operator to be used in streams with a different pacing than the predicted stream.

4 Future Offsets

The future offset operator enables references to stream values that will be available at a fixed position in the future relative to the current evaluation point. To illustrate the operator, consider the LTL formula from the introduction:

$$\Box(\text{takeoff} \Rightarrow \bigcirc(\neg\text{takeoff} \mathcal{U} \text{alt_reached})).$$

Intuitively, the monitor ensures that after each takeoff, the drone eventually reaches a valid flight altitude before the next takeoff occurs. A corresponding specification can be expressed in RTLola with future offsets as depicted in Figure 4, closely following the structure of the LTL formula. The stream `takeoff` detects takeoff events based on an altitude threshold and evaluates to true when the altitude is initially below the bound and exceeds it in the next time step. The stream `valid_flight` captures whether the current flight (until the next takeoff event) satisfies the altitude requirement. The trigger then ensures that every takeoff event corresponds to a valid flight.

An example is illustrated in Figure 5. The plot shows an altitude trace of a drone performing two consecutive flights. The color of the plotted points encodes the value of `valid_flight`. The triangles below the plot mark the takeoff events detected by the monitor. Their color corresponds to the evaluation of the trigger condition: On the first flight, the target altitude is not reached, so the trigger issues a warning. On the second flight, the altitude is reached successfully.

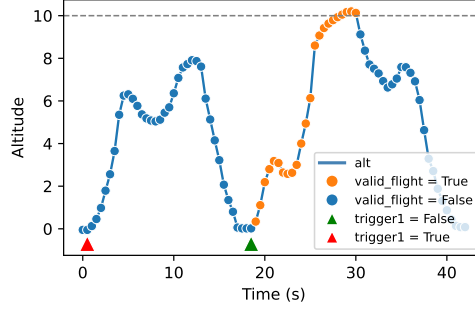


Fig. 5: The altitude monitoring of two consecutive flight of a drone.

Note that the violation is reported with the takeoff timestamp, allowing it to be associated with the correct flight that caused it.

In the remainder of this section, we first introduce real-time offsets in RT-LOLA, which naturally arise in future-oriented specifications, and show how they can be reduced to discrete offsets. We then formalize the semantics and adapt the static analyses to the extended offset operator.

4.1 Real-Time Offsets

In contrast to the prediction operator, offset accesses refer only to concrete stream values. The offset operator therefore operates over discrete positions in the stream, accessing the n -th value in the past or in the future. However, many specifications are more naturally expressed using real-time offsets, such as "the value two seconds ago" or "the value in five seconds".

In general, such references are not directly supported, since the monitor cannot guarantee that a value exists at an exact real-time position during execution. However, if the pacing information of streams provides sufficient knowledge about their temporal structure, a real-time offset can be translated into an equivalent discrete offset. We therefore introduce real-time offsets not as an operator with a new semantics, but rather as syntactic sugar for discrete offsets.

Definition 7 (Rewriting Real-Time Offsets). *Let s be a stream with periodic pacing δ_s , and let $t \in \mathbb{R}$ be a real-time duration. If there exists $k \in \mathbb{N}$ such that $t = k \cdot \delta_s$, then the following rewrite rule applies:*

$$s.\text{offset}_{RT}(by : t) = s.\text{offset}(by : k).$$

If no such k exists, the monitor cannot guarantee that the accessed value exists. Such expressions are rejected during static analysis.

4.2 Future-Offset Semantics

We refer to RTLola extended with the future offset operator as RTLola^f , and extend RTLola's semantics from [4] with two inference rules for future offsets:

$$\text{FO-HIT} \frac{o > 0 \quad \text{sf}x = \text{Sfx}(\omega, s, t) \quad \frac{|sfx| > o \quad \text{sf}x[o] = v}{s.\text{offset}(\text{by} : o) \Downarrow_{\omega}^t v}}{\quad} \quad \text{FO-MISS} \frac{o > 0 \quad \text{sf}x = \text{Sfx}(\omega, s, t) \quad |sfx| \leq o}{s.\text{offset}(\text{by} : o) \Downarrow_{\omega}^t \perp}$$

These rules are analogous to their past-offset counterparts (where $o < 0$), but operate on the suffix of the trace rather than its prefix. Specifically, $\text{Sfx}(\omega, s, t)$ denotes the sequence of all values of stream s at time t and beyond, retaining only positions where $\omega(s)(t) \neq \perp$. FO-HIT applies when the suffix is long enough to contain index o , yielding the value v found there. FO-MISS applies when the suffix is too short and returns $\perp$, which must be resolved by a default expression.

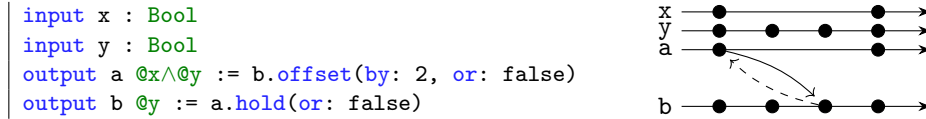
4.3 Well-Formedness

The addition of future offsets to RTLola^f introduces a new source of cyclic dependencies that must be considered for well-formedness.

Definition 8 (Well-Formedness). A RTLola^f specification φ is called *well-formed*, if the dependency graph G_{φ} does not contain any cycle with:

1. accumulated edge weight of zero.
2. a *Filter*-edge and an *Offset*-edge.
3. an positive *Offset*-edge and either an *Aggregate*-edge or a *Hold*-edge.

The first criterion originates from LOLA [14] and excludes synchronous cyclic dependencies. The second criterion was introduced for RTLola [27] to account for the interaction between filtering and offset accesses. The third criterion is specific to RTLola^f and addresses asynchronous cycles involving future offsets. It is best illustrated with an example. We use `s.offset(by: o, or: d)` as syntactic sugar for the expression `s.offset(by: o).defaults(to: d)`:



On the right, we depict one possible *evaluation graph* of the specification. The evaluation graph represents the concrete dependencies between stream instances at particular evaluation points. Since the timing of input streams x and y is not in control of the monitor, it cannot statically predict when a and b will be evaluated. In the example on the right, you can see one possible evaluation. However, in this case, the dashed `hold` access from b to a forms an evaluation cycle. The specification is therefore not well-defined, even though the corresponding cycle

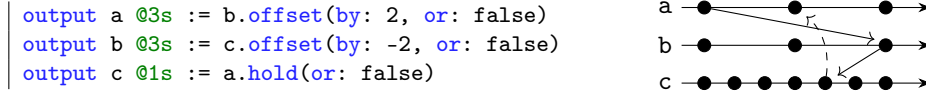
in the dependency graph contains a positive weight. The third well-formedness criterion excludes precisely such situations.

The well-formedness conditions exclude dependency patterns that may lead to unresolved cyclic dependencies during evaluation. This allows us to establish the central correctness result of the analysis.

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

The proof is given in Section B.

Periodic Dependency Cycles. For cycles that consist solely of periodic streams, the analysis can be less restrictive. Consider the following example:



Although the specification includes a cycle with an accumulated edge weight of zero, and therefore does not satisfy well-formedness according to Definition 8, it does not induce a cycle in the evaluation graph and is well-defined. At every evaluation point, all stream values can be computed without cyclic dependencies. For such specifications, we provide a transformation that uses the periodic pacing of the streams in the cycle. It assigns all streams a common pacing corresponding to the greatest common divisor of their original periods and adjusts the offset accesses accordingly. After this step, the resulting dependency cycle becomes synchronous and no longer forms a zero-cycle. You can find more details about this transformation in Section C.

4.4 Monitoring Algorithm

Algorithm 1 presents the monitoring algorithm for RTLOLA^f. The algorithm maintains three sets. R holds the resolved stream values that have been computed. U tracks timepoints at which stream values will definitely be computed, but for which the stream expression cannot be evaluated yet because of missing dependencies. Finally, M contains timepoints that *might* be evaluated, depending on whether the pacing condition is satisfied at that timepoint.

The algorithm is similar to that of LOLA [14], with the addition of the set M . The fixpoint loop alternates between pacing and value evaluation using the function *eval*. In each iteration, every pair currently contained in M (respectively U) is evaluated exactly once in arbitrary order, based on a snapshot of the set taken at the start of that iteration. For pacing, *eval* is applied to φ_s^p , the combined pacing and when-expression of stream s , and returns a boolean. For value computation, it is applied to φ_s , the with-expression of s . In both cases, *eval* may return ? (unresolved) when not all required dependencies are available yet.

The handling of offset accesses in *eval* deserves particular attention. To evaluate an offset expression, the correct accessed value needs to be found in R . The

Algorithm 1: Asynchronous Monitoring Algorithm

Input: specification φ , input stream assignment $I : \text{Time} \rightarrow (\text{ID}^\uparrow \rightarrow \mathbb{V}_\perp)$, time map $T : \text{Time} \rightarrow \mathbb{R}^+$

```

1  $R \leftarrow \emptyset \subseteq \mathcal{P}((\text{ID}^\uparrow \uplus \text{ID}^\downarrow) \times \mathbb{V}_\perp \times \text{Time})$            /* Resolved values */
2  $U \leftarrow \emptyset \subseteq \mathcal{P}(\text{ID}^\downarrow \times \text{Time})$                        /* Unresolved outputs */
3  $M \leftarrow \emptyset \subseteq \mathcal{P}(\text{ID}^\downarrow \times \text{Time})$                      /* Maybe outputs */
4 for  $t_M \in 1, 2, \dots, T_{\max}$  do
5    $R \leftarrow R \cup \{(s, I(t_M)(s), t) \mid s \in \text{ID}^\uparrow, I(t_M)(s) \neq \perp\}$ 
6    $M \leftarrow M \cup \{(s, t_M) \mid s \in \text{ID}^\downarrow\}$ 
7   while some set in  $\{R, U, M\}$  has changed do
8     foreach  $(s, t) \in M$  do
9        $M \leftarrow M \setminus \{(s, t)\}$ 
10      switch  $\text{eval}(\varphi_s^p, R, U, M, t_M, t, T)$  do
11        case ? do  $M \leftarrow M \cup \{(s, t)\}$ 
12        case true do  $U \leftarrow U \cup \{(s, t)\}$ 
13        case false do skip
14      foreach  $(s, t) \in U$  do
15         $U \leftarrow U \setminus \{(s, t)\}$ 
16        switch  $\text{eval}(\varphi_s, R, U, M, t_M, t, T)$  do
17          case ? do  $U \leftarrow U \cup \{(s, t)\}$ 
18          case v do  $R \leftarrow R \cup \{(s, t)\}$ 

```

algorithm resolves this by searching through R and U forwards or backward in time, until the desired offset is found. If found in R , the corresponding value is returned. However, if an entry in M is encountered along the way, the evaluation returns unresolved because it can not be decided which value will be accessed.

A further difference from LOLA is that this algorithm does not partially evaluate expressions. Instead, all values accessed by an expression are retained in memory until the expression can be evaluated in full. This has the advantage that if a stream value is required by multiple expressions, it needs to be stored only once, rather than duplicated across partial evaluation states.

To relate Algorithm 1 to the semantics of RTLola^f, we establish a correspondence between the values in R and in the evaluation model.

Theorem 2 (Correctness of Monitoring Algorithm). *Given a well-defined specification φ , an input stream assignment I , and a time map T . Let $\omega \in \llbracket \varphi \rrbracket$ be the unique evaluation model for the given input streams and time map, and let R_i denote the set R after the completion of iteration i of the algorithm. Then, for every $i \in \text{Time}$, the following holds:*

1. *If $(s, v, t) \in R_i$, then $\omega(s)(t) = v$.*
2. *If $\omega(s)(t) = v$, then there exists $i' \geq i$ such that $(s, v, t) \in R_{i'}$.*

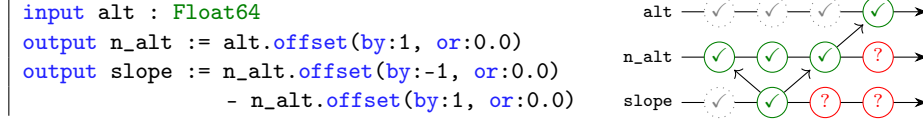


Fig. 6: Example specification with corresponding evaluation graph.

The theorem establishes that the monitoring algorithm is both sound and complete with respect to the semantics of RTLola^f . Soundness ensures that every value reported by the monitor algorithm corresponds to the value in the evaluation model, while completeness guarantees that all values in the evaluation model are eventually produced by the algorithm. The proof can be found in Section D.

4.5 Memory Analysis

The memory bound of a stream specifies how many past values must be retained to ensure the correct evaluation of all stream expressions in a specification. This bound may be unbounded, as is the case for the specification in Figure 4, because we can not statically know how long between two takeoff events. A specification with a finite memory bound is called *efficiently monitorable*, a term originally introduced by LOLA [14]. In contrast to LOLA, our algorithm does not partially evaluate unresolved expressions. While LOLA can discard intermediate results once they are inlined in the expression, we retain all accessed values until a stream expression can be fully evaluated.

We illustrate this with the example in Figure 6. The figure on the right illustrates an evaluation graph where unresolved values are shown in red, and resolved values are marked with a checkmark. Green nodes indicate values that are still required for future evaluations, whereas gray nodes are no longer needed and can be discarded. From this graph, the memory bound can be read directly: for `alt`, the bound is 1 since only the current value must be retained. For `n_alt` the bound is 4, as the fourth last value is still required to evaluate `slope`. This is because the execution of `slope` is delayed until all dependencies are resolved.

This delay is captured by the *delay* measure [16], which assigns to each stream the maximal temporal displacement induced by its dependencies. Intuitively, the delay $\Delta(s)$ of a stream s measures how far into the future the monitor must look to compute a new value for s . It equals the largest total offset accumulated along any path originating at s in the dependency graph. The delay becomes infinite when any such path passes through an asynchronous access to a stream that itself has a non-zero delay, since the waiting time is then unbounded.

Based on this notion, we define the memory bound of a stream.

Definition 9 (Memory Bound). Let φ be a well-defined RTLola^f specification with the corresponding dependency graph $G_\varphi = \langle V, E \rangle$. For a stream s , the

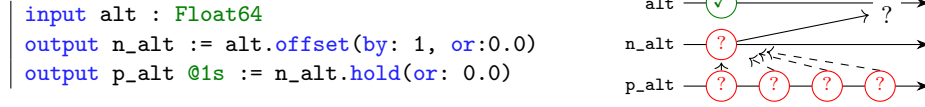


Fig. 7: Example for an unbounded specification without dependency cycles.

memory bound $\mu(s)$ is defined as:

$$\mu_\varphi(s) = \max \{ \Delta(s), \max \{ \Delta(s') - k \mid (s', _, k), s \in E \} \} + 1.$$

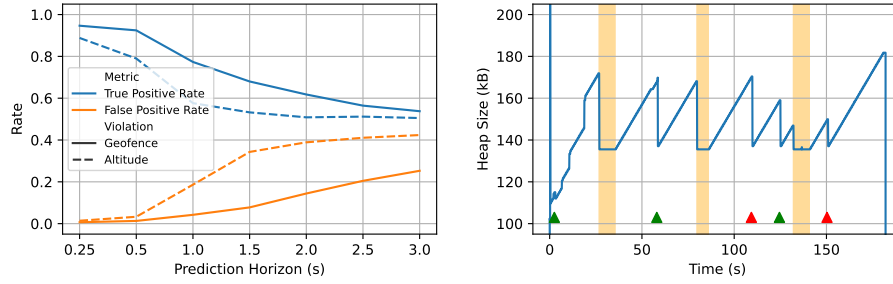
The following theorem shows that this bound is sufficient for the monitoring algorithm and therefore characterizes the amount of memory that must be retained for each stream. The proof can be found in Section F.

Theorem 3 (Memory Bound Soundness). *Let φ be a well-defined $RTLola^f$ specification and let μ_φ be its memory-bounds. During the execution of Algorithm 1, for every stream s and every value $(s, v, t) \in R$, if there exist $\mu(s)$ newer values of s in R , then (s, v, t) can be removed from R without affecting the correctness of the algorithm.*

Note that, in contrast to LOLA, unbounded memory requirements do not necessarily arise from dependency cycles. They can also occur due to unresolved asynchronous accesses. Consider the specification in Figure 7: The stream `n_alt` depends on the next value of `alt` and remains unresolved until that value arrives. The periodic stream `p_alt` accesses `n_alt` asynchronously, and therefore cannot be resolved until the next altitude reading arrives. Since the arrival time of the next input value is not known statically, the monitor cannot determine how many evaluations of `p_alt` will occur before `n_alt` becomes available.

4.6 Evaluation Order

The monitoring algorithm in Algorithm 1 operates as a fixpoint computation, iterating until no more values can be resolved. In practice, a well-chosen order can avoid unnecessary iterations. An evaluation order already existed for RTLola [27] where the dependency graph induces a topological order on streams. However, the original analysis considered only zero-offset edges, since past offset values are already computed in previous iterations. Future offsets break this assumption: a stream with a future offset access can only be evaluated once the referenced future value has been resolved. By accounting for the delay Δ of each stream, the dependency graph can be adjusted so that future offset accesses resemble synchronous zero-offset dependencies, from which a valid stream evaluation order is obtained by topological sort. Within each iteration, timepoints are processed in descending order: once a future value is resolved, all earlier stream instances that depend on it are resolved in the same pass, drastically reducing the number of fixpoint iterations. For the specification in Figure 6 this yields the stream order `alt` $\prec$ `n_alt` $\prec$ `slope`.



(a) True and false positive rates over prediction horizon. (b) Memory usage for unbounded specification.

5 Evaluation

In this section, we evaluate our implementation based on the RTLOLA StreamIR Interpreter [5]. All results were obtained on a system with a 13th Gen Intel Core i7-1355U, and the corresponding specifications are provided in Section G.

Prediction. To evaluate the prediction operator, we make use of a specification from [7], which involved monitoring the altitude and geofence bounds of a drone. Here, we evaluate the true positive rate (TPR) and false positive rate (FPR) when predicting trigger violations for different prediction horizons. For each prediction horizon, we obtain a predicted trigger verdict by extrapolating the monitored value across the horizon and checking it against the threshold, and compare it to the actual trigger verdict once that time is reached. All results use a fixed observation window of 5 values and a linear prediction method, evaluated across the 10 drone flight traces from [7] recorded in the AirSim simulator.

Figure 8a shows the TPR and FPR for both violation types across prediction horizons ranging from 0.5 to 3.0 seconds. At 0.25 seconds, the geofence monitor achieves a TPR of 94.7% at an FPR of 0.7%, while the altitude monitor yields a TPR of 88% at an FPR of 1.4%. As expected, performance degrades with increasing horizon, with the altitude being affected more strongly than the geofence. These results demonstrate that the prediction operator can detect violations more than one second in advance with a high accuracy, which is significant for cyber-physical systems operating on a millisecond timescale.

Bounded Specifications. To evaluate bounded specifications, we monitor a geofence recovery requirement: whenever the system exits the geofence, it must correct itself and re-enter the geofence within 2s. This obligation is most naturally expressed using a future real-time offset, which directly mirrors the forward-looking deadline in the requirement:

```
output @1Hz outside_fence : Bool := ...
trigger ¬(outside_fence ⇒ ¬outside_fence.offset(by: 2s, or: false))
```


The same property can equivalently be expressed using only past-time offsets, at the cost of inverting the reading:

| `trigger` $\neg(\text{outside_fence.offset}(\text{by: } -2\text{s, or: false}) \Rightarrow \neg\text{outside_fence})$

We evaluate the runtime overhead and memory consumption of both formulations by monitoring all traces consecutively across 200 runs. Figure 9 compares the runtime of both specifications for different offset values. While the runtime of the past-time specification remains essentially constant as the offset increases, the runtime of the future-time specification grows steadily. This is because unresolved values must be revisited until they can be resolved, causing the monitoring algorithm to process an increasing number of pending values.

Nevertheless, even at the largest evaluated offset, the future-time specification incurs only a 13.5% runtime overhead compared to its past-time counterpart, and without the cost of inverting the timepoints of the triggers. In contrast, memory consumption is nearly identical for both specifications and increases only marginally with larger offsets.

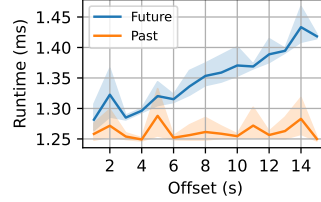


Fig. 9: Runtime comparison.

Unbounded Specifications. Finally, we evaluate the memory usage of monitors with unbounded memory. Figure 8b shows the memory usage for three drone flight traces from [7] while monitoring the specification from Figure 4. During monitoring, triangles at the bottom of the plot mark takeoff events, and orange-shaded regions indicate intervals during which the target altitude is reached. As visible in the figure, the memory resets whenever either a takeoff or an altitude reached event occurs. Since takeoff events occur at roughly known intervals, the memory consumption is effectively bounded in practice.

6 Conclusion

We have extended RTLola with two complementary mechanisms for reasoning about future stream values: a prediction operator and a future-offset operator. The prediction operator applies forecasting models to a sliding window of past observations, enabling lightweight, immediate estimates of future values. The future-offset operator provides exact access to future stream values at the cost of a delayed evaluation.

These two mechanisms are particularly natural in combination: predictions can be issued immediately, while future-offset accesses later confirm or refute them against the ground truth. Moreover, since predictions are inherently future-directed, it is natural to express the confirming specification in the same future-oriented style. This improves the expressiveness and readability of specifications, while introducing only minor computational overhead.

As future work, we plan to investigate the automatic rewriting of bounded future-offset specifications into equivalent past-time specifications. This would allow users to benefit from future-oriented specifications while avoiding their runtime overhead, by adapting evaluation timestamps such that verdicts are reported at the correct points in time.

Acknowledgments. This work was partially supported by the German Research Foundation (DFG) as part of TRR 248 (No. 389792660) and PreCePT (No. 521273327), and by the European Research Council (ERC) Grant HYPER (No. 101055412). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. J. Baumeister and F. Scheerer carried out this work as members of the Saarbrücken Graduate School of Computer Science.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Artale, A., Geatti, L., Gigante, N., Mazzullo, A., Montanari, A.: LTL over finite words can be exponentially more succinct than pure-past ltl, and vice versa. In: Artikis, A., Bruse, F., Hunsberger, L. (eds.) 30th International Symposium on Temporal Representation and Reasoning, TIME 2023, NCSR Demokritos, Athens, Greece, September 25-26, 2023. pp. 2:1–2:14. LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). <https://doi.org/10.4230/LIPICS.TIME.2023.2>
2. Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. *J. ACM* **62**(2), 15:1–15:45 (2015). <https://doi.org/10.1145/2699444>
3. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.* **20**(4), 14:1–14:64 (2011). <https://doi.org/10.1145/2000799.2000800>
4. Baumeister, J., Correnson, A., Finkbeiner, B., Scheerer, F.: An intermediate program representation for optimizing stream-based languages (2025), <https://arxiv.org/abs/2504.21458>
5. Baumeister, J., Correnson, A., Finkbeiner, B., Scheerer, F.: An intermediate program representation for optimizing stream-based languages. In: Piskac, R., Rákamaric, Z. (eds.) Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part III. Lecture Notes in Computer Science, vol. 15933, pp. 393–407. Springer (2025). https://doi.org/10.1007/978-3-031-98682-6_20
6. Baumeister, J., Finkbeiner, B., Kohn, F., Scheerer, F.: A tutorial on stream-based monitoring. In: Platzter, A., Rozier, K.Y., Pradella, M., Rossi, M. (eds.) Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part II. pp. 624–648. Lecture Notes in Computer Science, Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_33
7. Baumeister, J., Finkbeiner, B., Scheerer, F.: Active monitoring with RTLola: A specification-guided scheduling approach. In: Könighofer, B., Torfah, H. (eds.)

- Runtime Verification - 25th International Conference, RV 2025, Graz, Austria, September 15-19, 2025, Proceedings. pp. 181–201. Lecture Notes in Computer Science, Springer (2025). https://doi.org/10.1007/978-3-032-05435-7_11
8. Baumeister, J., Finkbeiner, B., Scheerer, F., Siber, J., Wagenpfeil, T.: Stream-based monitoring of algorithmic fairness. In: Gurfinkel, A., Heule, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part I. pp. 60–81. Lecture Notes in Computer Science, Springer (2025). https://doi.org/10.1007/978-3-031-90643-5_4
 9. Bozzelli, L., Sánchez, C.: Foundations of boolean stream runtime verification. In: Bonakdarpour, B., Smolka, S.A. (eds.) Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings. pp. 64–79. Lecture Notes in Computer Science, Springer (2014). https://doi.org/10.1007/978-3-319-11164-3_6
 10. Cairoli, F., Bortolussi, L., Paoletti, N.: Neural predictive monitoring under partial observability. In: Feng, L., Fisman, D. (eds.) Runtime Verification - 21st International Conference, RV 2021, Virtual Event, October 11-14, 2021, Proceedings. pp. 121–141. Lecture Notes in Computer Science, Springer (2021). https://doi.org/10.1007/978-3-030-88494-9_7
 11. Chen, Z., Ma, M., Li, T., Wang, H., Li, C.: Long sequence time-series forecasting with deep learning: A survey. *Inf. Fusion* **97**, 101819 (2023). <https://doi.org/10.1016/J.INFFUS.2023.101819>
 12. Chou, Y., Yoon, H., Sankaranarayanan, S.: Predictive runtime monitoring of vehicle models using bayesian estimation and reachability analysis. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2020, Las Vegas, NV, USA, October 24, 2020 - January 24, 2021. pp. 2111–2118. IEEE (2020). <https://doi.org/10.1109/IROS45743.2020.9340755>
 13. 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.) Formal Methods: Foundations and Applications - 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26-30, 2018, Proceedings. pp. 144–162. Lecture Notes in Computer Science, Springer (2018). https://doi.org/10.1007/978-3-030-03044-5_10
 14. 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 2005), 23-25 June 2005, Burlington, Vermont, USA. pp. 166–174. IEEE Computer Society (2005). <https://doi.org/10.1109/TIME.2005.26>
 15. Filipow, N., Main, E., Tanriver, G., Raywood, E., Davies, G., Douglas, H., Lavery, A., Stanojevic, S.: Exploring flexible polynomial regression as a method to align routine clinical outcomes with daily data capture through remote technologies. *BMC Med Res Methodol* (2023). <https://doi.org/10.1186/s12874-023-01942-4>
 16. Finkbeiner, B., Oswald, S., Passing, N., Schwenger, M.: Verified rust monitors for lola specifications. In: Deshmukh, J., Nickovic, D. (eds.) Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings. pp. 431–450. Lecture Notes in Computer Science, Springer (2020). https://doi.org/10.1007/978-3-030-60508-7_24

17. Gorostiaga, F., Sánchez, C.: Striver: Stream runtime verification for real-time event-streams. In: Colombo, C., Leucker, M. (eds.) *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*. pp. 282–298. *Lecture Notes in Computer Science*, Springer (2018). https://doi.org/10.1007/978-3-030-03769-7_16
18. Gorostiaga, F., Sánchez, C.: Stream runtime verification of real-time event streams with the striver language. *Int. J. Softw. Tools Technol. Transf.* **23**(2), 157–183 (2021). <https://doi.org/10.1007/S10009-021-00605-3>
19. Havelund, K., Rosu, G.: Synthesizing monitors for safety properties. In: Katoen, J., Stevens, P. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. pp. 342–356. *Lecture Notes in Computer Science*, Springer (2002). https://doi.org/10.1007/3-540-46002-0_24
20. Hipler, R., Kallwies, H., Leucker, M., Sánchez, C.: General anticipatory runtime verification. In: Gurfinkel, A., Ganesh, V. (eds.) *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II. Lecture Notes in Computer Science*, vol. 14682, pp. 133–155. Springer (2024). https://doi.org/10.1007/978-3-031-65630-9_7
21. Leucker, M.: Sliding between model checking and runtime verification. In: Qadeer, S., Tasiran, S. (eds.) *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*. pp. 82–87. *Lecture Notes in Computer Science*, Springer (2012). https://doi.org/10.1007/978-3-642-35632-2_10
22. Lichtenstein, O., Pnueli, A., Zuck, L.D.: The glory of the past. In: Parikh, R. (ed.) *Logics of Programs, Conference, Brooklyn College, New York, NY, USA, June 17-19, 1985, Proceedings*. pp. 196–218. *Lecture Notes in Computer Science*, Springer (1985). https://doi.org/10.1007/3-540-15648-8_16
23. Lindemann, L., Qin, X., Deshmukh, J.V., Pappas, G.J.: Conformal prediction for STL runtime verification. In: Mitra, S., Venkatasubramanian, N., Dubey, A., Feng, L., Ghasemi, M., Sprinkle, J. (eds.) *Proceedings of the ACM/IEEE 14th International Conference on Cyber-Physical Systems, ICCPS 2023, (with CPS-IoT Week 2023), San Antonio, TX, USA, May 9-12, 2023*. pp. 142–153. ACM (2023). <https://doi.org/10.1145/3576841.3585927>
24. Raszyk, M., Basin, D.A., Krstic, S., Traytel, D.: Multi-head monitoring of metric temporal logic. In: Chen, Y., Cheng, C., Esparza, J. (eds.) *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*. pp. 151–170. *Lecture Notes in Computer Science*, Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_9
25. Scheffel, T.: Expressiveness and complexity of stream-based specification languages. Ph.D. thesis, University of Lübeck, Germany (2022), <https://www.zhb.uni-luebeck.de/epubs/ediss2525.pdf>
26. Schumann, J., Moosbrugger, P., Rozier, K.Y.: R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. In: Bartocci, E., Majumdar, R. (eds.) *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015, Proceedings*. pp. 233–249. *Lecture Notes in Computer Science*, Springer (2015). https://doi.org/10.1007/978-3-319-23820-3_15
27. Schwenger, M.: Statically-analyzed stream monitoring for cyber-physical Systems. Ph.D. thesis, Saarland University, Saarbrücken, Germany (2022), <https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/33890>

28. Zhang, X., Leucker, M., Dong, W.: Runtime verification with predictive semantics. In: Goodloe, A., Person, S. (eds.) NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings. pp. 418–432. Lecture Notes in Computer Science, Springer (2012). https://doi.org/10.1007/978-3-642-28891-3_37

A Evaluation Graph

Definition 10 (Evaluation Graph). *Given a specification φ and an input trace I , the evaluation graph $EG_\varphi(I)$ is the least pair (V, E) satisfying:*

- $V \subseteq ID^\uparrow \times Time$, where $(sid, t) \in V$ iff
 - t satisfies the pacing constraint of sid , and
 - the *when*-condition of sid at time t evaluates to $\top$ using only values of stream instances $(sid', t') \in V$.
- $E \subseteq V \times V$, where $((sid, t), (sid', t')) \in E$ iff $(sid', t') \in V$ and the evaluation of sid' at time t' accesses the value of sid at time t (according to the RTLOLA expression semantics).

If no such pair (V, E) exists, then $EG_\varphi(I)$ is undefined.

Lemma 1. *If for every input trace i the evaluation graph $EG_\varphi(I)$ exists and is acyclic, then φ is well-defined.*

Proof. Let I be an arbitrary input trace. We show that there exists a unique evaluation model ω satisfying the stream-equations of φ . Since $EG_\varphi(I)$ exists, the set of evaluated stream instances is well-defined. Note that since $EG_\varphi(I)$ is acyclic, it admits a topological ordering $\prec$ on V .

- **Existence.** We define ω recursively along $\prec$:
 - **Base Case:** For (sid, t) with $sid \in ID^\uparrow$, set $\omega(sid)(t) = I(sid)(t)$.
 - **Recursive Case:** For (sid, t) with $sid \in ID^\downarrow$, all dependencies (sid', t') of (sid, t) satisfy $(sid', t') \prec (sid, t)$, so $\omega(sid')(t')$ is already defined. We set $\omega(sid)(t)$ by evaluating the expression sid at t using the already evaluated values.
- **Uniqueness.** Suppose ω, ω' are two evaluation models for I . We show $\omega = \omega'$ by induction along $\prec$.
 - **Base Case:** For $sid \in ID^\uparrow$, both models must satisfy $\omega(sid)(t) = I(sid)(t)$.
 - **Induction Step:** For (sid, t) with $sid \in ID^\downarrow$, all dependencies $(sid', t') \prec (sid, t)$ satisfy $\omega(sid')(t') = \omega'(sid')(t')$ by induction hypothesis. Since the expression of sid is a pure function of its dependencies, $\omega(sid)(t) = \omega'(sid)(t)$.

B Proof for Theorem 1

Proof. We prove the contraposition: If φ is not well-defined, then φ is not well-formed.

Assume φ is not well-defined. By the contraposition of Lemma 1, there exists an input trace I such that the evaluation graph $EG_\varphi(I)$ either does not exist or contains a cycle.

- $EG_\varphi(I)$ **does not exist.** The evaluation graph is undefined if it is not possible to construct the vertex set V satisfying all pacing and **when**-conditions. If the dependency relation induced by these conditions were acyclic, then V could be constructed in a topological order. Therefore, the non-existence of V implies that the dependency relation is cyclic. Since only **when**-conditions determine whether a vertex (sid, t) exists, any such cycle must contain a Filter-edge. This contradicts the well-formedness criterion (2). containing Filter-edges.
- $EG_\varphi(I)$ **contains a cycle.** Let

$$C = (sid_1, t_1) \rightarrow (sid_2, t_2) \rightarrow \dots \rightarrow (sid_n, t_n) \rightarrow (sid_1, t_1)$$

be a cycle in $EG_\varphi(I)$. We define $\Delta_{C,i} = t_{i+1} - t_i$ for edge i in C . Since C is a cycle:

$$\sum_{i=1}^n \Delta_{C,i} = 0$$

Note that if edge i is a

- discrete offset o : $\Delta_{C,i} = o \in \mathbb{Z}$,
- hold access: $\Delta_{C,i} \leq 0$, since it accesses the latest value at or before the current time,
- aggregate: $\Delta_{C,i} \leq 0$, since it accesses values in a past window.

Further, note that we count a synchronous access as one with an offset of 0.

We distinguish two cases:

- **Every Hold and Aggregate Edge in C has $\Delta_{C,i} = 0$.**

$$\sum_{i=1}^n \Delta_{C,i} = \sum_{\text{edge } i \text{ is offset}} o_i + \sum_{\text{edge } i \text{ is hold/agg}} 0 = 0$$

In this case, C induces a cycle in the dependency graph with accumulated weight 0. This violates the well-formedness condition (1), so φ is not well-formed.

- **There exists Hold or Aggregate in C with $\Delta_{C,i} < 0$** Since all Hold and Aggregate edges satisfy $\Delta_{C,i} \leq 0$, and at least one is strictly negative, their total contribution is negative:

$$\sum_{\text{edge } i \text{ is hold/agg}} \Delta_{C,i} < 0.$$

Since the total sum of C is zero, the offset edges must compensate:

$$\sum_{\text{edge } i \text{ is offset}} o_i > 0.$$

Hence, there exists at least one positive offset edge k in C with $o_k > 0$. The cycle C therefore induces a cycle in the dependency graph containing a positive offset edge and either an aggregate or an asynchronous hold edge. This contradicts the well-formedness condition (3).

C Periodic Well-Formedness

Consider a RTLOLA^f specification Φ and some cycle $C = \{(s_1, op, s_2), \dots, (s_n, op, s_1)\}$ in its corresponding dependency graph.

First compute the greatest common divider among all pacings

$$gcd = GCD(Pace(s_1), Pace(s_2), \dots, Pace(s_n))$$

Afterwards, transform the following edges:

```

output a @pa := b.offset(by: off) ⇒
output a @gcd := b.offset(by:  $\frac{pb \cdot off}{gcd}$ )

output a @pa := b.hold() ⇒
output a @gcd :=
  if Time mod pb ≡  $\frac{pb}{gcd} - 1$  then b.offset(by:  $-\frac{pb}{gcd} + 1$ ) else
  if Time mod pb ≡  $\frac{pb}{gcd} - 1 - gcd$  then b.offset(by:  $-\frac{pb}{gcd} + 1 + gcd$ ) else
  if Time mod pb ≡  $\frac{pb}{gcd} - 1 - 2gcd$  then b.offset(by:  $-\frac{pb}{gcd} + 1 + 2gcd$ ) else
  ...
  if Time mod pa ≡ 0 then b

output a @pa := b.aggregate(over: t, with: δ) ⇒
output a_values @gcd :=
  (b.offset(by:  $-\lceil \frac{t}{gcd} \rceil + 1$ ),
  b.offset(by:  $-\lceil \frac{t}{gcd} \rceil + 1 + gcd$ ),
  b.offset(by:  $-\lceil \frac{t}{gcd} \rceil + 1 + 2gcd$ ),
  ...
  b)
output a @gcd := δ(a_values)

```

Example 1. Given some RTLOLA^f specification:

```

output a @6s := b.offset(by: +1).defaults(to:0)
output b @3s := c.aggregate(over: 2s, using: sum)
output c @2s := a.hold(or: 0)

```

Now, we translate it according to the construction above:

```

output a @1s := b.offset(by:+3).defaults(to:0)
output b_values @1s := (c.offset(by:-1).defaults(to:0), b)
output b @1s := sum(b_values)
output c @1s := if Time mod 6s ≡ 5 then a.offset(by: -5) else
                  if Time mod 6s ≡ 4 then a.offset(by: -4) else
                  if Time mod 6s ≡ 3 then a.offset(by: -3) else
                  if Time mod 6s ≡ 2 then a.offset(by: -2) else
                  if Time mod 6s ≡ 1 then a.offset(by: -1) else
                  if Time mod 6s ≡ 0 then a

```

Hence, we obtain a desugared specification, which now contains a cycle with accumulated edge weight of zero. This means the specification is not well-formed.

D Algorithm Correctness

Definition 11 (Eval).

$$\begin{aligned}
 eval(sync(x), R, U, M, t_m, t, T) &= \begin{cases} v & \text{if } (x, v, t) \in R \\ ? & \text{otherwise} \end{cases} \\
 eval(offset(x, o), R, U, M, t_m, t, T) &= \begin{cases} eval(sync(x), \dots) & \text{if } o = 0 \\ \perp & \text{if } t \notin [0, T_{\max}] \\ ? & \text{if } t > t_m \vee (x, t) \in M \\ eval(offset(x, o'), \dots, t + 1) & \text{if } o > 0 \\ eval(offset(x, o'), \dots, t - 1) & \text{if } o < 0 \end{cases} \\
 eval(hold(x), R, U, M, t_m, t, T) &= \begin{cases} eval(sync(x), \dots) & \text{if } (x, _, t) \in R \\ eval(offset(x, -1), \dots) & \text{otherwise} \end{cases}
 \end{aligned}$$

with

$$o' = \begin{cases} o + 1 & \text{if } o < 0 \wedge hasEvent(x, t) \\ o - 1 & \text{if } o > 0 \wedge hasEvent(x, t) \\ o & \text{otherwise} \end{cases}$$

and

$$hasEvent(x, t) = (x, _, t) \in R \vee (x, t) \in U.$$

D.1 Proof for Theorem 2

Proof. We first argue that the while loop terminates in each iteration of the for loop. The sets M , U , and R are finite, and entries move monotonically: each entry can only move from M to U or be discarded, and from U to R . No entry can be re-added to M once removed.

Soundness. We show by induction on the order in which values are added to R that every $(sr, v, t) \in R$ satisfies $\omega(sr)(t) = v$. For the base case, input stream values are added directly from I in iteration t , which by definition agrees with ω . For the inductive step, assume all values currently in R are correct. A new output stream value (s, v, t) is added to R only when $eval(\varphi_s, R, U, M, t_m, t, T) = v$. By induction on the expression structure of φ_s , $eval$ returns a concrete value v only when all dependencies are resolved in R . By the induction hypothesis, all values in R are correct, so the returned value v must equal $\omega(s)(t)$ by the uniqueness of ω , which follows from the well-definedness of φ .

Completeness. We show that every value $\omega(sr)(t) = v$ eventually appears in R . For input streams, values are added in iteration t of the for loop directly from I . For output streams, every (s, t) is initially added to M at the start of iteration t . Since ω is a valid evaluation model, the pacing expression φ_s^p evaluates to *true* at every t where $\omega(s)(t) \neq \perp$. Once all dependencies of φ_s^p are

resolved in R , $eval(\varphi_s^p, \dots)$ returns *true*, moving (s, t) to U . Subsequently, once all dependencies of φ_s are resolved in R , $eval(\varphi_s, \dots)$ returns the concrete value $v = \omega(s)(t)$, moving (s, v, t) to R . By well-definedness of φ , all dependencies are eventually resolved across iterations of the for loop, so every output stream value eventually appears in R .

E Delay

Given some specification φ and its corresponding dependency graph G_φ . Let $weight(k)$ be the weight of some expression kind k , defined as:

$$weight(k) = \begin{cases} o & \text{if } k = (Offset, o) \\ 0 & \text{otherwise} \end{cases}$$

The delay $\Delta(s)$ of some stream s is given by:

$$\Delta(s) = \begin{cases} \infty & \text{if } \exists s' : (s, (_, Hold), s') \in E \\ & \wedge \Delta(s') > 0 \\ \max\{0, \max\{weight(k) + \Delta(s') \\ \mid (s, (_, k), s') \in E\}\} & \text{otherwise} \end{cases}$$

Theorem 4 (Delay Soundness). *For every stream s and timestep t , the evaluation of s at time t requires at most $\Delta(s)$ future values of any accessed stream to already be present in R .*

Proof. We distinguish two cases based on whether $\Delta(s)$ is finite or infinite.

Infinite delay: $\Delta(s) = \infty$ if s has a Hold edge to a stream s' with $\Delta(s') > 0$, or if s lies on a cycle with positive total weight. In the Hold case, we cannot bound how long it takes before a new resolved value of s' is produced.

Finite delay: If $\Delta(s) < \infty$, all cycles reachable from s have non-positive total weight. The required number of future values is therefore bounded by the maximum total weight along any path from s in G_φ , since non-positive cycles do not increase this accumulation. This maximum is exactly $\Delta(s)$ by definition.

F Proof for Theorem 3

Proof. We show that once there exist $\mu(s)$ newer values of s in R , no future call to *eval* will reference (s, v, t) again. A value (s, v, t) can only be accessed during the evaluation of some stream s' at event t' that has a direct dependency $(s', (_, k), s) \in E$, looking up the k -th next (or previous) event of s relative to t' . By definition of the delay, the evaluation of s' at event t' requires at most $\Delta(s')$ future events of any accessed stream to be present in R . Hence, (s, v, t) is no

longer needed by s' once there are more than $\Delta(s') - k$ newer values of s in R , since s' will then access a strictly newer event of s . Taking the maximum over all streams s' and offsets k with $(s', (_, k), s) \in E$, and adding one to convert the count into a buffer size, yields exactly $\mu_\varphi(s)$. Therefore, once $\mu(s)$ newer values of s exist in R , the value (s, v, t) can be safely removed without affecting any future evaluation.

G Specifications

G.1 Prediction

Example for prediction horizon of 3s.

```
import math
input gps_lat : Float64
input gps_lon : Float64
input barometer_pressure : Float64
input barometer_altitude : Float64
input gps_altitude : Float64
output start_lat := start_lat.offset(by:-1).defaults(to: gps_lat)
output start_lon := start_lon.offset(by:-1).defaults(to: gps_lon)
output start_altitude := start_altitude.offset(by:-1).defaults(to:
    gps_altitude)
output distance_to_start := sqrt((gps_lat-start_lat)*(gps_lat-start_lat)
    + (gps_lon-start_lon)*(gps_lon-start_lon))*10000.0
output altitude_above_ground := gps_altitude - start_altitude

output distance_to_start_timed @4Hz := distance_to_start.hold(or: 0.0)
output outside_geofence := distance_to_start_timed >= 8.0
output predicted_geofence @4Hz :=
    distance_to_start_timed.predict(using: linear, over_discrete: 5,
    in: 3.0s).defaults(to: 0.0) >= 8.0
output geofence_true_positive := outside_geofence &&
    predicted_geofence.offset(by: -3s).defaults(to: false)
output geofence_false_positive := !outside_geofence &&
    predicted_geofence.offset(by: -3s).defaults(to: false)
output geofence_true_negative := !outside_geofence &&
    !predicted_geofence.offset(by: -3s).defaults(to: false)
output geofence_false_negative := outside_geofence &&
    !predicted_geofence.offset(by: -3s).defaults(to: false)

output altitude_above_ground_timed @4Hz :=
    altitude_above_ground.hold(or: 0.0)
output altitude_violation := altitude_above_ground_timed >= 10.0
output predicted_altitude_violation @4Hz :=
    altitude_above_ground_timed.predict(using: linear, over_discrete:
    5, in: 3.0s).defaults(to: 0.0) >= 10.0
output altitude_true_positive := altitude_violation &&
    predicted_altitude_violation.offset(by: -3s).defaults(to: false)
```

```

output altitude_false_positive := !altitude_violation &&
    predicted_altitude_violation.offset(by: -3s).defaults(to: false)
output altitude_true_negative := !altitude_violation &&
    !predicted_altitude_violation.offset(by: -3s).defaults(to: false)
output altitude_false_negative := altitude_violation &&
    !predicted_altitude_violation.offset(by: -3s).defaults(to: false)

```

G.2 Bounded Specification

```

import math
input gps_lat : Float64
input gps_lon : Float64
input barometer_pressure : Float64
input barometer_altitude : Float64
input gps_altitude : Float64

output start_lat := start_lat.offset(by:-1).defaults(to: gps_lat)
output start_lon := start_lon.offset(by:-1).defaults(to: gps_lon)
output start_altitude := start_altitude.offset(by:-1).defaults(to:
    gps_altitude)

output distance_to_start := sqrt((gps_lat-start_lat)*(gps_lat-start_lat)
    + (gps_lon-start_lon)*(gps_lon-start_lon))*10000.0
output altitude_above_ground := gps_altitude - start_altitude

output distance_to_start_timed @1Hz := distance_to_start.hold(or: 0.0)

output outside_geofence := distance_to_start_timed >= 8.0

trigger !(outside_geofence -> !outside_geofence.offset(by:
    x).defaults(to: false))

```