




Active Monitoring with RTLola: A Specification-Guided Scheduling Approach

Jan Baumeister^{}, Bernd Finkbeiner^{}, and Frederik Scheerer^{}

CISPA Helmholtz Center for Information Security,
Saarbrücken, Germany

`{jan.baumeister, finkbeiner, frederik.scheerer}@cispa.de`

Abstract. Stream-based monitoring is a well-established runtime verification approach which relates input streams, representing sensor readings from the monitored system, with output streams that capture filtered or aggregated results. In such approaches, the monitor is a passive external component that continuously receives sensor data from the system under observation. This setup assumes that the system dictates what data is sent and when, regardless of the monitor’s current needs. However, in many applications – particularly in resource-constrained environments like autonomous aircraft, where energy, size, or weight are limited – this can lead to inefficient use of communication resources. We propose making the monitor an active component that decides, based on its current internal state, which sensors to query and how often. This behavior is driven by scheduling annotations in the specification, which guide the dynamic allocation of bandwidth towards the most relevant data, thereby improving monitoring efficiency. We demonstrate our approach using the stream-based specification language RTLola and assess the performance by monitoring a specification from the aerospace domain. With equal bandwidth usage, our approach detects specification violations significantly sooner than monitors sampling all inputs at a fixed frequency.

Keywords: Stream-based Monitoring · Constraint-Based Scheduling · Real-time Properties

1 Introduction

Cyber-physical systems are increasingly prevalent, and many now operate fully autonomously in complex, real-world environments. These systems are often deployed in safety-critical domains, such as autonomous vehicles or drones, where incorrect behavior can lead to catastrophic outcomes. Runtime monitoring is a well-established technique for checking the system’s behavior at runtime against a formal specification and, therefore, ensuring its correct behavior [14,18,22].

A prominent class of runtime monitoring frameworks is stream-based monitoring [6,21,16]. There, the system continuously supplies a monitor with data about its current state via input streams. Output streams compute new values by aggregating and filtering the inputs, often leveraging temporal operators to

express rich, time-dependent specifications. This makes stream-based monitoring particularly suitable for complex safety requirements of cyber-physical systems.

However, in modern autonomous systems, monitors must process data from a diverse set of sources in real-time. For example, an autonomous vehicle or drone might rely on GPS, camera feeds, and many other sensors to assess its environment and maintain safety. Here, the monitor is a passive component, consuming the updates about the current state of the system under observation. A key challenge in such settings is limited bandwidth: the connection between sensors and the monitor cannot support arbitrarily high-frequency updates from all sources due to factors like energy consumption, physical size, or weight constraints. Consequently, it is not feasible to transmit all sensor data at its highest possible frequency. However, reducing the sampling frequency leads to increased latency in detecting critical events and dangerous situations, undermining the effectiveness of the monitor.

We propose that, since the monitor has a comprehensive view of all received data, it has a better understanding of which data is currently needed than the different sources individually. For instance, consider a drone equipped with both an altitude sensor and a camera. When the drone is landed, altitude readings are irrelevant, while the ground-level camera might be more important for monitoring the surrounding ground. Conversely, at higher altitudes, the camera data becomes less informative, whereas altitude measurements become more significant as the drone is approaching predefined upper altitude limits. We present an active monitoring approach in which the system adaptively queries the sensors based on the drone’s current altitude. This enables dynamic prioritization of sensor data, emphasizing ground-level visual input when near the ground and focusing on altitude measurements when high up in the sky – therefore making more effective use of the limited bandwidth between the sensors and the monitor.

Such an approach is especially interesting for applications that are designed for actively requesting individual sensors’ data. A prominent example is the OBD-II interface present in all modern vehicles, which must be explicitly queried for individual sensors of the car, instead of receiving a stream of new sensor data on its own.

Given the availability of well-established monitoring tools [6,21,24,25], we chose not to develop a new monitoring framework from scratch. Instead, our approach builds on existing monitoring tools by introducing a scheduling component that interfaces between the monitor and the sensors. This scheduler observes the current monitoring state and determines which sensors to query next. Once the selected data has been acquired, the scheduler forwards it to the underlying monitor to initiate the next monitoring cycle. This architecture allows our method to remain independent of the monitor implementation and be compatible with a wide range of existing monitoring infrastructures.

We demonstrate our approach through an implementation based on the stream-based monitoring language RTLola [6], which has previously been successfully applied to the monitoring of unmanned aircraft [5,8] or for increasing trust in automatic decision and prediction systems [7]. We extend RTLola by

introducing scheduling annotations that can be attached to individual streams in the specification. These annotations guide the scheduling process by indicating which inputs should be prioritized under certain conditions of the monitor. The annotated specification is then automatically transformed into a "regular" RTLola specification compatible with all existing RTLola implementations [3,15,9]. The schedule is embedded into this transformed specification through additional output streams, which are then interpreted by the scheduler to determine which inputs should be queried in the next cycle.

We evaluated our approach using data obtained through the Microsoft AirSim simulator. Our results demonstrate that the proposed method can significantly reduce bandwidth consumption without compromising monitoring quality, or, on the other hand, detect violations earlier compared to a fixed-frequency approach, while utilizing the same amount of bandwidth.

Contribution. To summarize, we make the following contributions:

- We define a formal semantics for stream-based specification languages that incorporate scheduling of streams,
- We extend RTLola with an annotation mechanism to express scheduling constraints directly within the specification,
- We present a translation from annotated specifications to regular ones, which integrate the scheduling information as additional streams, and
- We implement our approach on top of RTLola and evaluate it in an online setting using simulated drone data.

The remainder of this paper is structured as follows: In Section 2, we provide a background on the RTLola monitoring language and define formal semantics for stream-based specification languages. In Section 3, we extend these semantics to incorporate the scheduling mechanism. Section 4 introduces our active monitoring framework and discusses challenges specific to integrating scheduling into RTLola. Finally, we present our evaluation in Section 5.

1.1 Related Work

Many monitoring frameworks assume that every change in the system is observable by the monitor. However, this assumption does not hold in bandwidth-constrained environments, where the monitor must instead periodically sample the system state. Bonakdarpour et al. [10] propose a time-driven monitoring approach that samples the system at a fixed frequency to reduce overhead. This frequency is statically determined to guarantee that no violations are missed, but it treats all inputs equally throughout execution. Navabpour et al. [23] extend this work by making the sampling path-aware, allowing the frequency to vary over time. Stoller et al. [26] also aim to reduce monitoring overhead by sampling the system periodically and use Hidden Markov Models to estimate unobserved states between samples. In contrast to the previous approaches which sample the system, our setting allows explicitly querying individual sensor, giving us the potential to prioritize inputs according to their importance. Huang et al. [20] take

a different approach by dynamically adjusting the amount of monitoring to stay within a user-specified target overhead. Instead of sampling, they temporarily disable the monitoring of certain events when the limit is exceeded otherwise.

Our approach is designed for the stream-based specification language RTLola [6], a successor of the synchronous stream-based specification language Lola [12]. RTLola has been successfully applied to the monitoring of cyber-physical systems, such as unmanned aircraft [5,8]. However, our approach is also transferable to other asynchronous stream-based languages, including Tessa [21] and Striver [16]. While there exist stream-based approaches that utilize annotations in specifications for correctness guarantees [4,17], we use annotations to represent timing constraints.

Another relevant area of research is scheduling in real-time systems, assigning tasks to processors – a topic comprehensively overviewed by Buttazzo [11] – and various approaches also consider bandwidth constraints (e.g., [27,13,1,2]). In this paper, we adapted these concepts to stream-based languages, by defining the scheduling semantics in the context of stream-based monitoring and integrating them into the specification language. While scheduling of real-time systems defines dependencies between tasks, stream-based settings provide a different kind of dependency constraints, relating jointly evaluated tasks. Our approach is closest to non-preemptive scheduling with dynamic priorities and hard aperiodic deadlines.

2 RTLola

Stream-based monitors operate over a set of *streams*, each representing an infinite sequence of values. RTLola specifications define these streams through stream-equations, which describe how the values in the sequences are computed. We distinguish between *input streams*, which are populated with data from the monitored system, and *output streams*, which compute new values by aggregating and filtering the inputs.

Consider the following example of an RTLola specification:

```

1 | input alt : Float64
2 | output alt_diff
3 |   eval @alt with abs(alt - alt.offset(by:-1).defaults(to: 0.0))
4 | trigger alt_diff > 10.0

```

The specification defines one input stream `alt` which is automatically populated with new readings from the drone’s altitude sensor as they arrive at the monitor. Next, the specification defines the output stream `alt_diff`, which computes the absolute difference between two consecutive altitude readings. This is achieved using the offset operator, which allows access to past stream values. Because such previous values are unavailable at the startup of the monitor, a default value must be provided that is used instead in this case. Finally, a trigger is defined to check whether the altitude difference exceeds 10. If the expression evaluates to true, the trigger activates, indicating a violation of the specification.

2.1 Types

Each stream in RTLola has two associated types: a value type and a pacing type. The *value type* specifies the kind of data contained in the stream, such as integers, floating-point numbers, or booleans. The *pacing type* defines *when* new values of a stream are computed and is indicated using the \mathbb{Q} -symbol following the `eval` keyword. In this paper, all output streams are *event-driven*, meaning they are evaluated whenever new inputs arrive. For example, the `alt_diff` stream always computes a new value whenever the `alt` stream receives a new value, as indicated by its pacing type.

Next, consider the following extension of the previous specification:

```

1 | output num_high_alt
2 |   eval @alt when alt > 20.0
3 |   with num_high_alt.offset(by:-1).defaults(to: 0) + 1

```

This stream is evaluated whenever the `alt` input stream receives a new value. Here, the evaluation is further conditioned by the dynamic filter condition provided after the `when` keyword: A new value is computed only if the condition evaluates to true. In this case, the stream counts how often a high-altitude reading occurs. RTLola also allows multiple `eval` clauses per stream. Clauses are checked from top to bottom, and the stream is evaluated using the `with` expression of the first clause whose `when` condition is satisfied.

For a more detailed explanation of the RTLola monitoring language – including features such as time-driven streams, aggregations, and parametrization – we refer the reader to the RTLola Tutorial [6].

2.2 Semantics

We use the following semantics for stream-based monitors. Although the formulation is applied to RTLola, the semantics are transferable to accommodate other stream-based specification languages. We define the semantics via an *evaluation model* $\omega \in \mathbb{W}$, which assigns stream references ID , consisting of input stream references ID^\uparrow and output stream references ID^\downarrow , to a timed series of values. We write $\omega(t)$ to refer to the real-time timestamp at discrete timestep $t \in \text{Time}$, and $\omega(sid)(t)$ to refer to the value of the stream *sid* at time t . If this stream did not calculate a new value at that time because of its pacing or filter conditions, \perp is returned instead.

The semantics ensure that each output stream value in the evaluation model is correctly computed according to the specification’s defining equations. Formally, given an RTLola specification φ , we define its semantics as the set:

$$\llbracket \varphi \rrbracket = \{ \omega \in \mathbb{W} \mid \forall sid \in ID^\downarrow. \forall t \in \text{Time}. \varphi(sid) \Downarrow_w^t \omega(sid)(t) \wedge \forall t \in \text{Time}. \omega(t) < \omega(t+1) \},$$

In the formula, $\varphi(sid) \Downarrow_w^t v$ denotes that the defining stream equations $\varphi(sid)$ of stream *sid* evaluate to v at time t . This evaluation yields the result of the `with`-expression of the first `eval` clause whose pacing and `when` condition is satisfied at time t . If no clause is evaluated, the result is \perp .

An evaluation model is considered valid with respect to a specification if, at every time step, all output stream values are correctly computed according to the defining stream equations and the time map is strictly monotonically increasing.

An RTLola specification φ is considered *well-defined*, if for every possible input trace $I \in \text{Time} \rightarrow \text{InputValues}$ with $\text{InputValues} : \text{ID}^\uparrow \rightarrow \mathbb{V}_\perp$, there exists a unique evaluation model $\omega \in \llbracket \varphi \rrbracket$ with $\forall t \in \text{Time}. \forall i \in \text{ID}^\uparrow. \omega(i)(t) = I(t)(i)$.

3 Scheduled Monitor Semantics

This section introduces the general concept for stream-based scheduling, where a scheduler must dynamically adapt to the current state of the monitor. We describe a valid schedule with a set of static constraints and later determine whether a scheduler satisfies them. We start by defining dynamic schedule constraints to decide at each time point if the evaluation violates the constraint. Then, we describe how to transform a static schedule into its dynamic counterpart – the transformation later implemented by the scheduling component.

In our setting, the scheduler determines at each time step which tasks are evaluated in the next step. The set of possible tasks defines the space of scheduling decisions available to the scheduler at each time step.

Definition 1 (Task). *For each $\tau \in \text{Tasks}$ there exists: 1. a predicate $\omega \models_t \tau$ to determine if the evaluation model $\omega \in \mathbb{W}$ satisfies the task τ at time t , 2. a partial order \preceq representing dependencies between tasks, and 3. a predicate $iv \models \mathcal{T}$ to determine if an input $iv : \text{InputValues}$ reflects the set of tasks \mathcal{T} .*

To illustrate different choices for the task space, consider two examples.

Example 1 (Individual Streams). For this example, consider a scheduler that selects individual streams during evaluation. Then $\text{Tasks} = \text{ID}$ consisting of all stream's references and

$$\begin{aligned} \omega \models_t \text{sref} & \quad \text{iff} \quad \omega(\text{sref})(t) \neq \perp \\ \tau_1 \preceq \tau_2 & \quad \text{iff} \quad \tau_1 = \tau_2 \\ iv \models \mathcal{T} & \quad \text{iff} \quad \forall i \in \text{ID}^\uparrow. iv(i) \neq \perp \Leftrightarrow \exists \text{sref} \in \mathcal{T}. \text{sref} = i. \end{aligned}$$

In this case, tasks are satisfied at time t whenever their stream receives a value, i.e., is not \perp at time t . There exist no dependencies between tasks, and an input reflects a set of tasks if each input receives a new value iff it is contained in the set of tasks.

Example 2 (Stream Sets). Alternatively, consider a scheduler selecting groups of streams to be jointly evaluated. We then set $\text{Tasks} = \mathcal{P}(\text{ID})$ and

$$\begin{aligned} \omega \models_t \tau & \quad \text{iff} \quad \forall \text{sref} \in \tau. \omega(\text{sref})(t) \neq \perp \\ \tau_1 \preceq \tau_2 & \quad \text{iff} \quad \tau_1 \subseteq \tau_2 \\ iv \models \mathcal{T} & \quad \text{iff} \quad \forall i \in \text{ID}^\uparrow. iv(i) \neq \perp \Leftrightarrow \exists \tau \in \mathcal{T}. i \in \tau. \end{aligned}$$

In this representation, tasks have dependencies through a subset relation, and an input reflects a set of tasks if at least one task updates each input.

3.1 Dynamic Schedule Constraints

Next, we introduce the concept of dynamic scheduling constraints, which specify how each task is expected to be evaluated:

Definition 2 (Dynamic Schedule Constraint). A dynamic schedule constraint is a function that maps the current monitor state $(\mathbb{W}, \text{Time})$ to a scheduling decision over tasks:

$$S : (\mathbb{W} \times \text{Time}) \rightarrow \text{Tasks} \rightarrow \mathbb{S} \quad \mathbb{S} = \{Y, M, N\}$$

For each task, the constraint assigns whether the task must (Y), may (M), or must not (N) be evaluated at that time.

The semantics of stream-based languages ensure that all stream values are computed in accordance with their defining stream equations. We extend this semantics to account for scheduling constraints, requiring the evaluation model to also satisfy the tasks according to a constraint. In addition, we introduce a *bandwidth bound* B , which encodes the bandwidth limitations between sensors and the monitor. Formally, given a specification φ , a schedule constraint $\psi \in S$, and a bandwidth constraint B , the *scheduled semantics* is defined as:

$$\llbracket (\varphi, \psi, B) \rrbracket = \{\omega \in \mathbb{W} \mid \omega \in \llbracket \varphi \rrbracket \wedge \omega \in \llbracket \psi \rrbracket \wedge \forall t \in \text{Time}. B(\omega, t)\}.$$

Intuitively, an evaluation model is valid if it: 1. correctly computes all stream values according to φ , 2. adheres to the scheduling decisions made by ψ , and 3. respects the bandwidth constraint B at all times.

For an evaluation model to satisfy a constraint $\psi \in S$, the satisfaction or non-satisfaction of tasks must always align with ψ :

$$\llbracket \psi \rrbracket = \left\{ \omega \in \mathbb{W} \mid \forall t \in \text{Time}. \forall \tau \in \text{Tasks}. \begin{cases} \omega \models_{t+1} \tau & \text{if } \psi(\omega, t)(\tau) = Y \\ \top & \text{if } \psi(\omega, t)(\tau) = M \\ \omega \not\models_{t+1} \tau & \text{if } \psi(\omega, t)(\tau) = N \end{cases} \right\}.$$

If the schedule constraints assign Y to τ at time t , the evaluation model must satisfy that task at time $t+1$. If the decision is N, it must not satisfy that task, while for M, both outcomes are permitted.

Last, we define the bandwidth constraints imposed by the communication between sensors and the monitor. These constraints are formalized as a predicate $B : (\mathbb{W} \times \text{Time}) \rightarrow \mathbb{B}$, which determines whether the inputs received by the monitor at time t conform to the bandwidth limitations. In this paper, we consider a simple constraint model InputEventBound_b that limits the number of input streams that can receive a value at the same time to a fixed threshold b :

$$\text{InputEventBound}_b(\omega, t) = |\{i \in \text{ID}^\uparrow \mid \omega(i)(t) \neq \perp\}| \leq b.$$

Other constraints could account for the varying bit widths of individual input types or enforce protocols that require specific combinations of inputs.

3.2 Static Scheduling Constraints

This section introduces three *static schedule constraints* – deadlines, priorities, and their combination – and presents a translation deriving their dynamic counterpart. Static schedule constraints contain *conditions* Cond , which can be evaluated to a boolean under a given evaluation state $\Downarrow_t^\omega : \text{Cond} \rightarrow \mathbb{B}$, to constrain a task differently at runtime. In the following paragraphs, we describe these static schedules individually.

Deadline The first static schedule constraints describe deadlines. The constraints assign upper bounds to tasks, which indicate that a task should not be evaluated later than its deadline. It is defined as

$$\text{Deadline} : \text{Tasks} \rightarrow \mathcal{P}(\text{Cond} \times \mathbb{R})$$

and assigns each task to a set of pairs, each consisting of conditions and a corresponding deadline. Given such a static schedule constraint $\xi \in \text{Deadline}$, we derive a dynamic schedule $\psi \in S$ as follows:

$$\psi(\omega, t)(\tau) = \begin{cases} Y & \text{if } \exists (c, dl) \in \xi(\tau) \wedge \exists t' < t. c \Downarrow_{t'}^\omega \top \\ & \wedge \forall t'' \in (t', t]. \omega \not\models_{t''} \tau \\ & \wedge \omega(t+2) > \omega(t') + dl \\ M & \text{otherwise} \end{cases}$$

Intuitively, assuming tasks represent individual streams, a stream must be evaluated at time $t+1$ (i.e., $\psi(\omega, t)(\tau) = Y$) if there exists a condition in the static schedule that was satisfied at an earlier time t' and no new value has been produced for that stream since. Then, $t+1$ is the last chance to produce a value for that stream before violating its deadline. Otherwise, the scheduling decision defaults to M, allowing the stream's evaluation to be postponed.

Priority We may want to fully utilize the available bandwidth without manually assigning explicit deadlines. To support this use case, we introduce the *priority*-based static schedule constraint. In this case, each task is assigned a priority depending on conditions, dynamically determining its importance:

$$\text{Priority} : \text{Tasks} \rightarrow \mathcal{P}(\text{Cond} \times \mathbb{N}).$$

At each time step, the scheduler must select the streams with the highest priority. Given a static schedule constraint $\xi \in \text{Priority}$, we determine the current priority of a task with Prio_ξ :

$$\begin{aligned} \text{Prio}_\xi : \text{Tasks} \times \mathbb{W} \times \text{Time} &\rightarrow (\text{Cond} \times \mathbb{N})_\perp \\ \text{Prio}_\xi(\tau, \omega, t) &= \begin{cases} \arg \max_{(c,p) \in S} (\max\{t' \mid t' \leq t \wedge c \Downarrow_{t'}^\omega \top\}) & \text{if } (c, p) \text{ exists} \\ \perp & \text{otherwise} \end{cases} \end{aligned}$$

$Prio_\xi$ returns the most recent priority assignment of a task (if any) whose condition evaluates to true in ω at some time point $t' \leq t$. Using this function, we translate a static schedule $\xi \in \text{Priority}$ into a dynamic schedule $\psi \in S$ with

$$\psi(\omega, t)(\tau) = \begin{cases} \text{Y} & \text{if } (c_1, p_1) = Prio_\xi(\tau, \omega, t) \\ & \wedge \exists \tau'. (c_2, p_2) = Prio_\xi(\tau', \omega, t) \\ & \wedge \omega \models_{t+1} \tau' \wedge p_1 > p_2 \\ & \wedge \neg \exists \tau'' \succeq \tau. \omega \models_{t+1} \tau'' \\ \text{M} & \text{otherwise} \end{cases}$$

With this definition, a task must be selected if another, lower-priority task is selected for evaluation at the next step. This restriction, however, does not hold if the task is part of a larger, high-priority task, indicated by the dependency relation.

Deadline and Priority The priority schedule allows specifying the relative importance of tasks without explicitly reasoning about individual deadlines. However, this can lead to starvation: a lower-priority task may never be evaluated if higher-priority tasks continuously occupy all available bandwidth. To address this issue, we propose a combined schedule that merges priorities with deadlines. By default, the schedule behaves like the priority schedule. However, each τ is assigned a deadline dl_τ , which defines the maximum duration it may remain unevaluated before it is considered *overdue*:

$$\begin{aligned} overdue &: \text{Tasks} \times \mathbb{W} \times \text{Time} \rightarrow \mathbb{B} \\ overdue(\tau, \omega, t) &= \exists \tau' \subseteq \tau. \\ &\quad \omega(t) - \omega(\max\{t' \in \text{Time} \mid t' < t \wedge \omega \models_{t'} \tau'\}) > dl_\tau \end{aligned}$$

An overdue stream should be evaluated, regardless of the assigned priority. We translate a static schedule constraint $\xi \in \text{Priority}$ into a dynamic schedule constraint $\psi \in S$ with:

$$\psi(\omega, t)(\tau) = \begin{cases} \text{Y} & \text{if } \exists \tau'. overdue(\tau, \omega, t+1) \wedge \neg overdue(\tau', \omega, t+1) \\ & \wedge \omega \models_{t+1} \tau' \\ \text{Y} & \text{if } (c_1, p_1) = Prio_s(\tau, \omega, t) \\ & \wedge \exists \tau'. (c_2, p_2) = Prio_s(\tau', \omega, t) \\ & \wedge \omega \models_{t+1} \tau' \wedge p_1 > p_2 \wedge \neg overdue(\tau', \omega, t+1) \\ & \wedge \neg \exists \tau'' \succeq \tau. \omega \models_{t+1} \tau'' \\ \text{M} & \text{otherwise} \end{cases}$$

This definition assigns overdue tasks an even higher priority, but in general, it follows the previous definition.

3.3 Valid Scheduler

A scheduler is a program that, given schedule and bandwidth constraints, decides at each time point which set of tasks to evaluate. If all selections respect the constraints, the scheduler is considered valid:

Definition 3 (Valid Scheduler). *Given a static schedule constraint ξ over a set of Tasks, a specification φ and a bandwidth bound B , a scheduler $SA_{\xi,B} : \mathbb{W} \times \text{Time} \rightarrow \mathcal{P}(\text{Tasks})$ that decides at each timepoint which streams are evaluated at the next time step, is valid if*

1. $\forall \omega \in \llbracket (\varphi, \xi, B) \rrbracket. \forall t \in \text{Time}. \forall \mathcal{T} \subseteq \text{Tasks}. \forall i \in \text{InputValues}.$
 $\text{prefix}_{SA_{\xi,B}}(\omega, t) \wedge SA_{\xi,B}(\omega, t) = \mathcal{T} \wedge i \models \mathcal{T}$
 $\rightarrow \exists \omega' \in \llbracket (\varphi, \xi, B) \rrbracket. \text{validTasks}(\omega', t+1, \mathcal{T}) \wedge \omega'[..t] = \omega[..t] \wedge \omega'[t+1] = i.$
2. $\forall \omega \in \llbracket (\varphi, \xi, B) \rrbracket. \forall t \in \text{Time}. \exists t' > t. |SA_{\psi,B}(\omega, t')| \geq 1$

with

$$\begin{aligned}
 &\text{validTasks} : \mathbb{W} \times \text{Time} \times \mathcal{P}(\text{Tasks}) \rightarrow \mathbb{B} \\
 &\text{validTasks}(\omega, t, \mathcal{T}) = \forall \tau \in \mathcal{T}. \omega \models_t \tau \wedge \forall \tau \notin \mathcal{T}. \omega \not\models_t \tau \\
 &\quad \wedge \forall \tau_1, \tau_2 \in \text{Tasks}. \tau_1 \preceq \tau_2 \wedge \tau_2 \in \mathcal{T} \rightarrow \tau_1 \in \mathcal{T} \\
 &\text{prefix}_{SA_{\xi,B}} : \mathbb{W} \times \text{Time} \rightarrow \mathbb{B} \\
 &\text{prefix}_{SA_{\xi,B}}(\omega, t) = \forall t' < t. \text{validTasks}(\omega, t' + 1, SA_{\xi,B}(\omega, t'))
 \end{aligned}$$

A scheduler is considered valid if it satisfies two properties: 1. The scheduler must never get stuck. That is, for every point in time and any possible choice of input values consistent with the selected tasks, the evaluation model must be able to continue as a correctly scheduled evaluation model. 2. The scheduler may not indefinitely select empty tasks. These conditions ensure that the scheduler defines a sound evaluation strategy.

4 Active Scheduling in RTLola

This section demonstrates how our scheduling approach is integrated with the stream-based monitoring language RTLola. The overall architecture is illustrated in Figure 1. First, we allow users to augment RTLola specifications with scheduling annotations, as shown on the left side of the figure. These annotations assign deadlines or priorities to individual streams or clauses, and are detailed in Section 4.1. A translator then processes the annotated specification and produces a transformed RTLola specification. During this step, the translator interprets the scheduling annotations and adds helper streams that encode the scheduling constraints. We describe the translation process in Section 4.2 in more detail. The resulting specification is compatible with existing RTLola implementations, while a separate scheduling component interfaces between the backend and the sensors to issue sensor queries at runtime. We further explain this scheduler interface in Section 4.3.

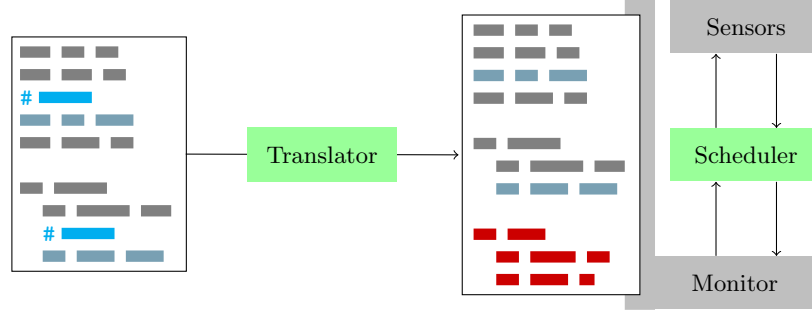


Fig. 1: Overview of the scheduling process.

4.1 Annotations

For the configuration of the scheduler, we embed the scheduling-related information as annotations directly within the specification. These annotations attach constraints to streams – either directly on input streams or on the `eval`-clause of output streams. Each annotation defines either a *priority* or a *deadline* – following the idea from Section 3.2. We represent the annotations as a mapping $a \in \text{ID} \rightarrow \mathcal{P}(\text{Cond} \times \mathbb{V})$. The conditions are a combination of the pacing of the stream and an expression derived from the `when`-conditions of the annotated eval clause. However, since `when`-conditions are evaluated top-to-bottom, the effective condition for each clause includes not only its own `when`-condition but also the negation of all preceding ones. This ensures that conditions are mutually exclusive, reflecting the semantics of sequential `when` evaluation. Depending on the kind of static schedule used for scheduling, the value \mathbb{V} is either a deadline $\in \mathbb{R}$ or a priority $\in \mathbb{N}$. For deadline-priority scheduling, annotations consist of priorities, while deadlines of individual tasks are configured globally.

Example 3. The specification in Figure 2 monitors two conditions: 1. whether the current latitude and longitude violate a geofence, and 2. whether the current altitude exceeds an upper bound. The annotations result in the following representation:

$$\begin{aligned}
 s(\text{bound_violation}) &= \{(\text{@lat\&\&lon}, \text{distance_to_bound} < 12.0), 10\}, \\
 &\quad (\text{@lat\&\&lon}, \text{distance_to_bound} \geq 12.0), 1\}, \\
 s(\text{altitude_violation}) &= \{(\text{@alt}, \text{true}), 5\}
 \end{aligned}$$

The specification assigns the output `bound_violation` a high priority, numerically represented as 10, if the distance to the bound is smaller than 12. If it is further away from the bound, the bound check is assigned a low priority, a 1. The altitude check is assigned a constant medium priority, represented as a 5. As a result, when the system is far away from a geofence boundary, the altitude has a higher priority and is therefore checked more frequently. Conversely, as the system approaches a potential geofence violation, geofence monitoring is prioritized.

```

1 | input lat, lon, alt : Float64
2 | output distance_to_bound
3 |   eval @lat&&lon
4 |     with min(lat - 3, UPPER_LAT - 8, lon - 5, UPPER_LON - 10)
5 | output bound_violation
6 |   #[priority="high"]
7 |   eval @lat&&lon when distance_to_bound < 12.0
8 |     with distance_to_bound < 0.0
9 |   #[priority="low"]
10 |  eval @lat&&lon when distance_to_bound >= 12.0
11 |    with false
12 | output altitude_violation
13 |   #[priority="medium"]
14 |   eval @alt with altitude > 50.0

```

Fig. 2: Example of scheduling annotations in RTLola.

4.2 Translation

A translator converts the annotated specification into a standard RTLola specification that any existing implementation can process. This translation aims to ensure that the resulting monitor satisfies the specified scheduling constraints by guiding the scheduling component to supply the necessary inputs at appropriate times. For this, it adds additional output streams to the resulting specification, which are in turn read by the scheduling component.

At first glance, one expects the scheduling annotations to correspond directly to the static schedule. However, setting `Tasks = ID` leads to an issue, as illustrated in Figure 3. Assume the scheduler is restricted to providing a new value to only one input stream at a time. In this scenario, there is no valid schedule that updates any inputs. If stream *a* receives a new value due to its high priority, stream *c* is also evaluated, triggered by its pacing. Yet, a stream with higher priority, namely *b*, is not evaluated, contradicting the scheduling semantics.

```

1 | #[priority="high"]
2 | input a : UInt64
3 | #[priority="medium"]
4 | input b : UInt64
5 | output c
6 |   #[priority="low"]
7 |   eval @a when true with a + 1

```

Fig. 3: Example specification.

The root of the issue is, that while output streams can be annotated with scheduling constraints, in practice, the evaluation of output streams is controlled by the underlying monitor and determined by its pacing type – the scheduler can only decide when to emit new input values to the monitor – and not prevent the evaluation of outputs. Consequently, the scheduler must supply inputs at a frequency that respects the scheduling annotations of all outputs streams, by satisfying their pacing at the right times. To support this idea, we restrict tasks to sets of input streams, i.e., $\text{Tasks} = \mathcal{P}(\text{ID}^\uparrow)$, and propagate all annotations from

output streams to the corresponding sets of inputs necessary for their evaluation.

Definition 4 (RTLola Tasks). *Given an RTLola specification φ containing input streams ID^\uparrow , we define the task set $\text{Tasks} = \mathcal{P}(\text{ID}^\uparrow)$ with*

$$\begin{aligned} \omega \models_t \tau & \quad \text{iff} \quad \forall i \in \tau. \omega(i)(t) \neq \perp \\ \tau_1 \preceq \tau_2 & \quad \text{iff} \quad \tau_1 \subseteq \tau_2 \\ i \models \mathcal{T} & \quad \text{iff} \quad \forall s \in \text{ID}^\uparrow. i(s) \neq \perp \Leftrightarrow \{i\} \in \mathcal{T}. \end{aligned}$$

Given annotations $a : \text{ID} \rightarrow \mathcal{P}(\text{Cond} \times \mathbb{V})$, we construct the static schedule ξ_a as the following, where $\text{pac}(s)$ returns the pacing of stream $s \in \text{ID}$:

$$\begin{aligned} \xi_a : \mathcal{P}(\text{ID}^\uparrow) & \rightarrow \mathcal{P}(\text{Cond} \times \mathbb{V}) \\ \xi_a(\tau) & = \{(c, v) \mid v = \max \{v' \mid \exists s \in \text{ID}. \text{pac}(s) \subseteq \tau \wedge (c', v') \in a(s) \wedge c' \Rightarrow c\}\} \end{aligned}$$

In other words, for each task – i.e., set of input streams – the schedule includes all constraint-condition pairs (c, v) such that v is the maximal value out of all annotations of streams whose pacing is implied by τ , and whose conditions imply c . The maximal value is chosen according to the domain \mathbb{V} : it represents the most restrictive constraint, e.g., the highest priority, or the shortest deadline.

Through pacing types in RTLola, the task dependency relation is not sufficient to determine if a task set is valid. Consider the following example where we have two input streams, i and i' , and a task set containing the tasks i and i' . When both input streams are updated, the pacing consisting of their combination also activates because of the RTLola semantics. Therefore, we need to ensure that the task containing the combination is also part of the task set. We therefore have to strengthen the notion of valid task sets introduced in Definition 3 with the additional constraint

$$\forall \tau_1, \tau_2 \in \text{Tasks}. \tau_1, \tau_2 \in \mathcal{T} \rightarrow \tau_1 \cup \tau_2 \in \mathcal{T}.$$

To represent this static schedule as additional RTLola streams, we construct a single stream for each task, i.e., each pacing type. We utilize different clauses guarded by the corresponding condition to dynamically assign the schedule value of a task based on the current state. The clauses are ordered by their schedule value in ascending order, so that more restrictive schedules are considered first. Further, for each task, separate output streams are added, which note the time of the task's last evaluation, and if needed, additional streams to indicate if a task is overdue. For the specification in Figure 2, the translation process would add the following output streams to the specification:

```

1 | output schedule_lat_lon
2 |   eval @lat&&lon when distance_to_bound < 10.0 with 10
3 |   eval @lat&&lon when distance_to_bound >= 10.0 with 1
4 | output last_lat_lon eval @lat&&lon with now
5 | output schedule_alt eval @alt with 5
6 | output last_alt eval @alt with now

```

4.3 Scheduler

While all previous steps occur before the monitor is executed, the scheduler is the runtime component responsible for querying new inputs and passing them to the underlying monitor. In each cycle, the scheduler queries sensors and forwards the obtained values to the monitor, triggering a new evaluation step. As usual, the monitor computes new output stream values, including the schedule streams that guide the scheduling decisions. The scheduler uses these newly computed values to decide which inputs to query in the next cycle.

The strategy the scheduler uses to populate the available bandwidth depends on the type of scheduling constraints. For all strategies, the bandwidth is specified as the number of input values per event b and the number of events per second f_e . At runtime, the scheduler emits events to the monitor at the specified frequency and populates each event with the predefined number of inputs according to the scheduling strategy.

For deadline scheduling, the scheduler $DS_{\xi,B}$ employs an earliest-deadline-first strategy [19]. In this mode, each task is associated with a deadline representing the maximum allowable timestamp of the next update. The scheduler tracks the current time and selects the task with the most urgent deadlines first until the bandwidth is fully utilized. In contrast, for priority scheduling, the scheduler $PS_{\xi,B}$ fills the event with tasks, selected in decreasing order of priority. It is the same for the deadline-priority scheduler $DPS_{\xi,B}$, only that overdue tasks are assigned to a new, highest priority. If multiple streams share the same priority level, the scheduler resolves this by choosing the stream that was updated the longest time ago. The formal definition of these schedulers can be found in Appendix B. With some restrictions, these schedulers satisfy the validity conditions in Definition 3:

Theorem 1 (Valid RTLola Schedulers). *Given a well-defined specification φ , an annotation $a \in \text{ID} \rightarrow \mathcal{P}(\text{Cond} \times \mathbb{V})$, and a bound $B = \text{InputEventBound}_b$:*

- *The schedulers $PS_{S_a,B}$ and $DPS_{S_a,B}$ are valid if $b \geq \max_{\tau \in \text{Tasks}} |\tau|$,*
- *The scheduler $DS_{\xi_a,B}$ is valid if $\forall \tau \in \text{Tasks}. \forall (c, dl) \in \xi(\tau). dl > n$ and $b \geq \max_{\tau \in \text{Tasks}} |\tau|$, with:*

$$n = \min\{n' \mid \forall \mathcal{T} \in \text{Permutations}_{|\text{Tasks}|}(\text{Tasks}). \text{splits}_B(\mathcal{T}, \varepsilon) = n'\}$$

$$\begin{aligned} \text{Permutations}_{|\text{Tasks}|}(\text{Tasks}) &= \left\{ \tau_1 \tau_2 \dots \in \text{Tasks}^{|\text{Tasks}|} \mid \forall i, j. i = j \vee \tau_i \neq \tau_j \right\} \\ \text{splits}_B(\varepsilon, \text{cur}) &= 1 \\ \text{splits}_B(\tau_1 \circ \mathcal{T}, \text{cur}) &= \begin{cases} 1 + \text{splits}_B(\tau_1 \circ \mathcal{T}, \varepsilon) & \text{if } |\text{cur} \cup \tau_1| > b \\ \text{splits}_B(\mathcal{T}, \text{cur} \cup \tau_1) & \text{otherwise} \end{cases} \end{aligned}$$

Generally, the bandwidth bound b needs to be large enough to accommodate each task individually. For the priority and deadline-priority schedulers, in each step, the tasks can be selected purely on the current priority assignments, and there

are no constraints from previous evaluations. The schedule selects the highest priority tasks. Our construction can trigger tasks with a potentially lower priority; however, the dependency relation, together with the construction through the scheduling annotations, ensures that these priorities do not conflict with the semantics. For the deadline scheduler, we need to account for constraints from previous cycles. There, the bound ensures that previously imposed deadlines can't conflict with newly added ones. Detailed proof sketches for the validity of each scheduler can be found in Appendix B.

5 Implementation & Evaluation

We have implemented our approach¹ from Section 4 on top of the RTLola monitoring framework [6]. For the evaluation, we use the AirSim simulator, a simulation environment for drones and cars developed by Microsoft. AirSim exposes various sensors through an API, including GPS coordinates, altitude readings, camera images, and LIDAR data.

For our evaluation, we investigate the following setting with the deadline-priority constraints. A drone is tasked to collect barometer data during flight for an experiment in a restricted airspace. It relies on four sensors, which must share the limited available bandwidth of the drone: the GPS and altitude sensor for enforcing their respective bounds, and two barometers, whose data collection for the experiment is the primary target of the flight. We use stream annotations as presented in Section 4.1 to represent different priorities for the tasks in the specification. The closer the drone comes to the geofence border, threatening a violation, the higher the assigned priority of the geofence task becomes. Likewise, the closer the drone comes to an altitude violation, the higher the assigned priority. The barometers, which serve the experiment, are assigned a constant medium priority. As a result, they receive most of the bandwidth if there is no immediate danger of a boundary violation. We employ overdue deadlines to ensure the geofence bounds are checked at least every 3 seconds to prevent violations from being missed because of higher priority tasks. The annotated specification can be found in Figure 4.

In our experiments, we compare the behavior of our scheduled monitoring approach against a set of baseline monitors that query all inputs at fixed frequencies. Each baseline monitor queries all sensors at the same frequency, but we vary this frequency across baselines to explore the trade-off between bandwidth and responsiveness. To ensure a fair comparison, all monitors observe the same drone flight in parallel.

The goal of the evaluation is to assess the effectiveness of actively scheduling the inputs in runtime monitoring under bandwidth constraints. Specifically, we aim to answer the following questions: 1. Can we reduce the overall bandwidth consumption while maintaining a comparable level of monitoring quality? 2. Can

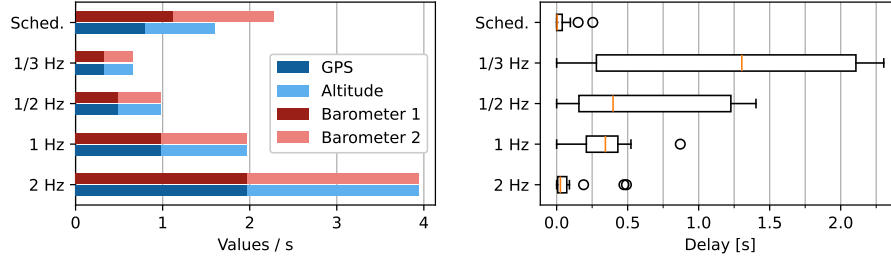
¹ The artifacts can be found on <https://github.com/reactive-systems/rtlola-active-monitoring>.

```

1  #![frequency="2Hz",bound="2"]
2  import math
3  #[deadline="3s"]
4  input gps_lat_long : (Float64, Float64)
5  #[deadline="3s"]
6  input gps_altitude : Float64
7  #[priority="medium",deadline="3s"]
8  input barometer_pressure : Float64
9  #[priority="medium",deadline="3s"]
10 input barometer_altitude : Float64
11 output lat := gps_lat_long.0
12 output long := gps_lat_long.1
13 output start_lat @gps_lat_long :=
    start_lat.offset(by:-1).defaults(to: lat)
14 output start_long @gps_lat_long :=
    start_long.offset(by:-1).defaults(to: long)
15 output start_altitude @gps_altitude :=
    start_altitude.offset(by:-1).defaults(to: gps_altitude)
16 output distance_to_start := sqrt((lat-start_lat)*(lat-start_lat)
17    + (long-start_long)*(long-start_long))*10000.0
18 output altitude_above_ground := gps_altitude - start_altitude
19 output geofence := distance_to_start ≥ 8.0
20 output scheduled_geofence
21    #[priority="low"]
22    eval @gps_lat_long when distance_to_start ≤ 4.0
23        with geofence
24    #[priority="medium"]
25    eval @gps_lat_long when distance_to_start ≤ 6.0
26        with geofence
27    #[priority="high"]
28    eval @gps_lat_long with geofence
29 trigger scheduled_geofence "outside geofence"
30 output altitude_bound := altitude_above_ground ≥ 10.0
31 output scheduled_altitude_bound
32    #[priority="low"]
33    eval @gps_altitude when altitude_above_ground ≤ 5.0
34        with altitude_bound
35    #[priority="medium"]
36    eval @gps_altitude when altitude_above_ground ≤ 7.5
37        with altitude_bound
38    #[priority="high"]
39    eval @gps_altitude with altitude_bound
40 trigger scheduled_altitude_bound "altitude too high"

```

Fig. 4: Evaluation Specification



(a) Bandwidth consumption of each sensor. (b) Delay until trigger is detected.

Fig. 5: Comparison between the scheduled monitor and four fixed-frequency ones.

we detect specification violations earlier by allocating the bandwidth more intelligently – focusing on data that is more likely to reveal a violation?

We evaluate the performance of our scheduling approach over 10 flights with the simulator by analyzing two metrics obtained from the evaluation. Figure 5a shows the average number of values that occupy the bandwidth for each sensor. For each monitor, we group the sensors into two categories, represented by the horizontal bars: The blue bar captures the bandwidth used for geofence and altitude checks, while the red bar shows the bandwidth used for collecting barometer data for the experiment. In the fixed-frequency monitors, bandwidth is distributed evenly across all sensors. In contrast, the scheduled monitor dynamically adjusts the frequency of sensor queries based on the current state of the system, resulting in an uneven allocation of bandwidth. As shown in the figure, the scheduled monitor allocates more bandwidth to the barometer sensors, aligning with the experiment’s objective of maximizing barometer data collection. Meanwhile, the sensors responsible for enforcing the geofence and altitude boundaries are queried less frequently. In contrast, Figure 5 presents a boxplot of the delay in detecting violations. Delays are measured relative to the earliest point in time any monitor detects the same violation.

To answer the first question, the results show that our scheduled monitor detects violations as early as the fastest fixed-frequency monitor operating at 2 Hz. Notably, it achieves this while consuming half the bandwidth, as evident from the comparison in the previous figure.

To answer the second question, we compare the scheduled monitor against a fixed-frequency monitor with equal bandwidth usage. The scheduled monitor consumes four values per second – equivalent to the bandwidth of a 1 Hz fixed-frequency monitor. Despite this, the scheduled monitor detects violations significantly earlier than the 1 Hz monitor, because in critical situations, the scheduled monitor focuses on the more critical inputs. This demonstrates that our approach makes more effective use of the available bandwidth through our scheduling approach.

6 Conclusion

We addressed the challenge of runtime monitoring in bandwidth-constrained environments by presenting an approach to dynamically allocating the available bandwidth to different sensors depending on the monitor’s current state. This approach enables the monitor to prioritize inputs and allocate bandwidth to data most likely to reveal specification violations. To facilitate this, we define a formal semantics for scheduled monitors and introduce several static schedules. Scheduling annotations, embedded directly within the specification, offer an intuitive and flexible mechanism to express constraints guiding the scheduler. A scheduler interfacing between the monitor and the sensors is responsible for querying sensors at the appropriate times and passing the values to the underlying monitor. We evaluated our approach using simulations in the AirSim drone simulator. The results demonstrate the effectiveness of scheduled monitoring in detecting violations early on, while consuming significantly less bandwidth compared to a monitor receiving all inputs at a fixed frequency.

For future work, we want to evaluate our approach in a real-world setting to assess the overhead introduced by querying sensors with the active role of the monitor. Furthermore, many real-world settings involve sensors producing multiple values, leading to more complex task dependencies. We want to investigate how our approach can be extended to handle such cases.

Acknowledgments. This work was partially supported by the German Research Foundation (DFG) as part of TRR 248 (No. 389792660) and by the European Research Council (ERC) Grant HYPER (No. 101055412).

References

1. Afshar, S., Behnam, M., Bril, R.J., Nolte, T.: Resource sharing under global scheduling with partial processor bandwidth. In: 10th IEEE International Symposium on Industrial Embedded Systems, SIES 2015, Siegen, Germany, June 8-10, 2015. pp. 195–206. IEEE (2015). <https://doi.org/10.1109/SIES.2015.7185061>
2. Agrawal, A.: Hardware Contention-Aware Real-Time Scheduling on Multi-Core Platforms in Safety-Critical Systems. Ph.D. thesis, Kaiserslautern University of Technology, Germany (2019), <https://kluedo.ub.rptu.de/frontdoor/index/index/docId/5612>
3. Baumeister, J., Correnson, A., Finkbeiner, B., Scheerer, F.: An intermediate program representation for optimizing stream-based languages. In: Piskac, R., Rakamarić, Z. (eds.) Computer Aided Verification. pp. 393–407. Springer Nature Switzerland, Cham (2025). https://doi.org/10.1007/978-3-031-98682-6_20
4. Baumeister, J., Dauer, J.C., Finkbeiner, B., Schirmer, S.: Monitoring with verified guarantees. *Int. J. Softw. Tools Technol. Transf.* **25**(4), 593–616 (2023). <https://doi.org/10.1007/S10009-023-00712-3>
5. Baumeister, J., Finkbeiner, B., Kohn, F., Löhr, F., Manfredi, G., Schirmer, S., Torrens, C.: Monitoring unmanned aircraft: Specification, integration, and lessons-learned. 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. 207–218. Springer (2024). https://doi.org/10.1007/978-3-031-65630-9_10
6. Baumeister, J., Finkbeiner, B., Kohn, F., Scheerer, F.: A tutorial on stream-based monitoring. In: International Symposium on Formal Methods. pp. 624–648. Springer (2024)
 7. 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. pp. 60–81. Springer Nature Switzerland, Cham (2025). https://doi.org/10.1007/978-3-031-90643-5_4
 8. Baumeister, J., Finkbeiner, B., Schirmer, S., Schwenger, M., Torens, C.: Rtlola cleared for take-off: Monitoring autonomous aircraft. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12225, pp. 28–39. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_3
 9. Baumeister, J., Finkbeiner, B., Schwenger, M., Torfah, H.: FPGA stream-monitoring of real-time properties. ACM Trans. Embed. Comput. Syst. **18**(5s), 88:1–88:24 (2019). <https://doi.org/10.1145/3358220>
 10. Bonakdarpour, B., Navabpour, S., Fischmeister, S.: Sampling-based runtime verification. In: Butler, M.J., Schulte, W. (eds.) FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6664, pp. 88–102. Springer (2011). https://doi.org/10.1007/978-3-642-21437-0_9
 11. Buttazzo, G.: Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Real-Time Systems Series, Springer US (2011)
 12. d’Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: Lola: runtime monitoring of synchronous systems. In: 12th International Symposium on Temporal Representation and Reasoning (TIME’05). pp. 166–174. IEEE (2005)
 13. Ereameev, A.V., Malakhov, A.A., Sakhno, M.A., Sosnovskaya, M.Y.: Multi-core processor scheduling with respect to data bus bandwidth. CoRR **abs/2010.16058** (2020), <https://arxiv.org/abs/2010.16058>
 14. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Broy, M., Peled, D.A., Kalus, G. (eds.) Engineering Dependable Software Systems, NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 34, pp. 141–175. IOS Press (2013). <https://doi.org/10.3233/978-1-61499-207-3-141>
 15. Faymonville, P., Finkbeiner, B., Schledjewski, M., Schwenger, M., Stenger, M., Tentrup, L., Torfah, H.: Streamlab: Stream-based monitoring of cyber-physical systems. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11561, pp. 421–431. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_24
 16. Gorostiaga, F., Sánchez, C.: Striver: Stream runtime verification for real-time event-streams. In: International Conference on Runtime Verification. pp. 282–298. Springer (2018)
 17. Hagen, G., Tinelli, C.: Scaling up the formal verification of lustre programs with smt-based techniques. In: Cimatti, A., Jones, R.B. (eds.) Formal Methods in

- Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008. pp. 1–9. IEEE (2008). <https://doi.org/10.1109/FMCAD.2008.ECP.19>
18. Havelund, K., Goldberg, A.: Verify your runs. In: Meyer, B., Woodcock, J. (eds.) *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*. Lecture Notes in Computer Science, vol. 4171, pp. 374–383. Springer (2005). https://doi.org/10.1007/978-3-540-69149-5_40
 19. Horn, W.: Some simple scheduling algorithms. *Naval Research Logistics Quarterly* **21**(1), 177–185 (1974)
 20. Huang, X., Seyster, J., Callanan, S., Dixit, K., Grosu, R., Smolka, S.A., Stoller, S.D., Zadok, E.: Software monitoring with controllable overhead. *Int. J. Softw. Tools Technol. Transf.* **14**(3), 327–347 (2012). <https://doi.org/10.1007/S10009-010-0184-4>
 21. Kallwies, H., Leucker, M., Schmitz, M., Schulz, A., Thoma, D., Weiss, A.: Tesla—an ecosystem for runtime verification. In: *International Conference on Runtime Verification*. pp. 314–324. Springer (2022)
 22. Leucker, M., Schallhart, C.: A brief account of runtime verification. *J. Log. Algebraic Methods Program.* **78**(5), 293–303 (2009). <https://doi.org/10.1016/J.JLAP.2008.08.004>
 23. Navabpour, S., Bonakdarpour, B., Fischmeister, S.: Path-aware time-triggered runtime verification. In: Qadeer, S., Tasiran, S. (eds.) *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*. Lecture Notes in Computer Science, vol. 7687, pp. 199–213. Springer (2012). https://doi.org/10.1007/978-3-642-35632-2_21
 24. Perez, I., Goodloe, A.E., Dedden, F.: Runtime verification in real-time with the copilot language: A tutorial. In: Platzer, 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*. Lecture Notes in Computer Science, vol. 14934, pp. 469–491. Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_27
 25. Rozier, K.Y., Schumann, J.: R2U2: tool overview. In: Reger, G., Havelund, K. (eds.) *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*, September 15, 2017, Seattle, WA, USA. Kalpa Publications in Computing, vol. 3, pp. 138–156. EasyChair (2017). <https://doi.org/10.29007/5PCH>
 26. Stoller, S.D., Bartocci, E., Seyster, J., Grosu, R., Havelund, K., Smolka, S.A., Zadok, E.: Runtime verification with state estimation. In: Khurshid, S., Sen, K. (eds.) *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*. Lecture Notes in Computer Science, vol. 7186, pp. 193–207. Springer (2011). https://doi.org/10.1007/978-3-642-29860-8_15
 27. Xu, D., Wu, C., Yew, P.: On mitigating memory bandwidth contention through bandwidth-aware scheduling. In: Salapura, V., Gschwind, M., Knoop, J. (eds.) *19th International Conference on Parallel Architectures and Compilation Techniques, PACT 2010, Vienna, Austria, September 11-15, 2010*. pp. 237–248. ACM (2010). <https://doi.org/10.1145/1854273.1854306>

A Semantics

Definition 5 (Evaluation Model).

$$\begin{aligned} \text{Stream} &: \text{Time} \rightarrow \mathbb{V}_\perp \\ \text{StreamMap} &: \text{ID} \rightarrow \text{Stream} \\ \text{TimeMap} &: \text{Time} \rightarrow \mathbb{R} \\ \mathbb{W} &: \text{StreamMap} \times \text{TimeMap} \end{aligned}$$

The evaluation model includes a mapping *StreamMap* that assigns each input and output stream reference to a stream – a sequence of values indexed by discrete time steps. At any time $t \in \text{Time}$, a stream may either contain a value or \perp , indicating that the stream does not contain a value at that time (e.g., due to pacing or a filter condition not being satisfied). The *TimeMap* assigns each discrete time step a real-valued timestamp.

For convenience, we use the following shorthand notation for $\omega = (\text{streams}, \text{times})$:

- $\omega(t) := \text{times}(t)$ gives the real-time timestamp at step $t \in \text{Time}$, and
- $\omega(\text{sid})(t) := \text{streams}(\text{sid})(t)$ gives the value of stream $\text{sid} \in \text{ID}$ at time $t \in \text{Time}$.

B RTLola Schedulers

A Scheduler is formally defines as:

Definition 6 (Scheduler). Given $\tau_i, \tau_j \in \text{Tasks}$, we define total orders

$$\begin{aligned} \tau_i \leq_{DL} \tau_j &\Leftrightarrow le(\tau_i) + v(\tau_i) \leq le(\tau_j) + v(\tau_j) \\ \tau_i \leq_P \tau_j &\Leftrightarrow \begin{cases} v(\tau_i) \leq v(\tau_j) & \text{if } v(\tau_i) \neq v(\tau_j) \\ le(\tau_i) \leq le(\tau_j) & \text{otherwise} \end{cases} \\ \tau_i \leq_{DP} \tau_j &\Leftrightarrow \begin{cases} od(\tau_i) \wedge \neg od(\tau_j) & \text{if } od(\tau_i) \neq od(\tau_j) \\ v(\tau_i) \leq v(\tau_j) & \text{if } od(\tau_i) = od(\tau_j) \\ & \wedge v(\tau_i) \neq v(\tau_j) \\ le(\tau_i) \leq le(\tau_j) & \text{otherwise} \end{cases} \end{aligned}$$

where $v(\tau)$, $le(\tau)$, and $od(\tau)$ are shorthand notations for stream-accesses to the helper streams holding the schedule value, the time of last evaluation and their overdue status respectively.

Given a schedule ξ , a bound B , and a total order \leq , we define

$$PS_{\xi, B, \leq} = \text{takeEvent}_B(\text{sort}_{\leq}(\text{Tasks}))$$

with

$$\text{takeEvent}_B(\mathcal{T}, \text{cur}) = \begin{cases} \text{takeEvent}_B(\text{tail}, \text{cur} \cup \text{task}_1) & \text{if } \mathcal{T} \tau_1 \circ \text{tail} \wedge |\text{cur} \cup \tau_1| < B \\ \{\tau \in \text{Tasks} \mid \tau \subseteq \text{cur}\} & \text{otherwise} \end{cases}$$

The schedulers are then defined as $DS_{\xi,B} = PS_{\xi,B,\leq_{DL}}$, $PS_{\xi,B} = PS_{\xi,B,\leq_P}$, and $DPS_{\xi,B} = PS_{\xi,B,\leq_{DP}}$.

Then, we can first proof that these schedulers only produce valid task sets:

Theorem 2 (Valid RTLola Task Sets). *Given a well-defined specification φ , an annotation mapping a , and a bound $b \geq \max_{\tau \in Tasks} |\tau|$ with $B = InputEventBound_b$, the schedulers $DS_{\xi_a,B}$, $PS_{\xi_a,B}$, and $DPS_{\xi_a,B}$ produce at every time point valid task sets.*

Proof. The proof follows directly from the construction of the task sets from the takeUntil definition and the relations \leq_{DL} , \leq_P , and \leq_{DP} .

To proof Theorem 1, we proof that each scheduler individually is valid, but first we introduce two side lemmas:

Lemma 1. *Given a well-defined specification φ , an evaluation model $\omega \in \llbracket \varphi \rrbracket$, an annotation mapping a , and a bound $B = InputEventBound_b$ with $b \geq \max_{\tau \in Tasks} |\tau|$, and $\forall \tau \in Tasks. \forall (c, dl) \in \xi_a(\tau). dl > n$ where*

$$n = \max\{n' \mid \forall \tau_1 \tau_2 \dots \in Permutations_{|Tasks|}(Tasks). splits_B(\tau_1 \tau_2 \dots, \varepsilon) = n'\}$$

. Then for every time $t \in Time$:

$$\begin{aligned} & (\forall t' < t. validTasks(\omega, t', DS_{\xi_a,B}(\omega, t'))) \\ & \rightarrow split'(sort_{\leq_{DL}}(Tasks)) = e_0 \dots e_m \wedge \forall \tau \in e_j. relativeDl_{\xi_a}(\tau, \omega, t) \geq j \end{aligned}$$

, with *relativeDl* returns the relative timepoint when the least deadline of this task is due and *split'*:

$$\begin{aligned} splits'_B(\varepsilon, cur) &= \begin{cases} \{\tau \in Tasks \mid \tau \subseteq cur\} \circ \epsilon & \text{if } cur \neq \emptyset \\ \epsilon & \text{otherwise} \end{cases} \\ splits'_B(\tau_0 \circ \mathcal{T}, cur) &= \begin{cases} \{\tau \in tail \mid \tau \subseteq cur\} \circ splits'_B(\tau_0 \circ tail, \varepsilon) & \text{if } |cur \cup \tau_0| > b \\ splits'_B(tail, cur \cup \tau_0) & \text{otherwise} \end{cases} \end{aligned}$$

Proof. We proof this lemma by induction over t :

$t=0$ Since $\forall \tau \in Tasks. \forall (c, dl) \in \xi_a(\tau). dl > n$, we know from the assumption with the minimal deadlines

$$\forall \tau \in Tasks. relativeDl_{\xi_a}(\tau, \omega, t) > n$$

since τ is either a new task, where the assumption holds or does not have yet a relative deadline. In addition, by construction of *split'*, *split'*(*sort* _{\leq_{DL}} (*Tasks*)) returns a sequence with a size of at most n . So $\tau \in e_j. relativeDl_{\xi_a}(\tau, \omega, t) > n \geq j$.

$t \rightarrow t + 1$ From the induction, we know at time t

$$split'(sort_{\leq_{DL}}(\text{Tasks})) = e_0 \dots e_m \wedge \forall \tau \in e_j.relativeDl_{S_a}(\tau, \omega, t) \geq j$$

Since ω follows the earliest deadline first algorithm, the inputs at time $t + 1$ follow e_0 , so $\forall \tau \in e_0.\omega \models_{t+1} \tau$. If we assume that at $t + 1$ no new deadlines are added, then at time $t + 1$:

$$split'(sort_{\leq_{DL}}(\text{Tasks})) = e_1 \dots e_m e_0$$

due to the ordering \leq_{DL} and by construction it follows directly:

$$\forall \tau \in e_j.relativeDl_{\xi_a}(\tau, \omega, t) \geq j$$

, since the relative deadline is reduced by one step but so also the position. If new deadlines are added at time $t + 1$, then

$$\forall \tau \in e_j.relativeDl_{\xi_a}(\tau, \omega, t) \geq \min(n, j) \geq j$$

, since either the task τ is assigned with a new deadline then $relativeDl_{\xi_a}(\tau, \omega, t) > n$ or the task keeps its old deadline. Then, this relative deadline is

- either smaller than n , then because of the ordering at time t , $\tau \in e_i$ and $relativeDl_{\xi_a}(\tau, \omega, t) \geq i$ and at time $t + a$ it holds $\tau \in e'_i = e_{i+1}$, so $relativeDl_{\xi_a}(\tau, \omega, t) \geq i$.
- greater or equal n , then $relativeDl_{\xi_a}(\tau, \omega, t) > n \geq i$

Lemma 2. *Given a well-defined specification φ , a world $\omega \in \llbracket \varphi \rrbracket$, an annotation mapping a , and a bound $B = \text{InputEventBound}_b$ with $b \geq \max_{\tau \in \text{Tasks}} |\tau|$, and $\forall \tau \in \text{Tasks}.\forall (c, dl) \in \xi_a(\tau).dl > n$ where*

$$n = \max\{n' \mid \forall \tau_1 \tau_2 \dots \in \text{Permutations}_{|\text{Tasks}|}(\text{Tasks}).splits_B(\tau_1 \tau_2 \dots, \varepsilon) = n'\}$$

. Then $\omega \in \llbracket \psi_{\xi_a} \rrbracket$ if $\forall t.validTasks(\omega, t, DS_{\xi_a, B}(\omega, t'))$.

Proof. For this proof, we unroll the definition of $\omega \in \llbracket \psi_{\xi_a} \rrbracket$, so we have to proof:

$$\forall t \in \text{Time}.\forall \tau \in \text{Tasks}.\begin{cases} \omega \models_{t+1} \tau & \text{if } \psi_{\xi_a}(\omega, t)(\tau) = Y \\ \top & \text{if } \psi_{\xi_a}(\omega, t)(\tau) = M \\ \omega \not\models_{t+1} \tau & \text{if } \psi_{\xi_a}(\omega, t)(\tau) = N \end{cases}$$

with

$$\psi_{S_a}(\omega, t)(\tau) = \begin{cases} Y & \text{if } \exists (c, dl) \in \xi_a(\tau) \wedge \exists t' < t.c \Downarrow_{t'}^\omega \top \\ & \wedge \forall t'' \in (t', t].\omega \not\models_{t''} \tau \wedge \omega(t+2) > \omega(t') + dl \\ M & \text{otherwise} \end{cases}$$

Given an arbitrary but fix t and τ , then

$$\psi_{\xi_a}(\omega, t)(\tau) = Y \rightarrow \omega \models_{t+1} \tau$$

$\psi_{\xi_a}(\omega, t)(\tau) = Y$ is by construction equivalent to $relativeDl_{\xi_a}(\tau, \omega, t) = 1$ and from Lemma 1, we know $\forall \tau \in e_j. relativeDl_{\xi_a}(\tau, \omega, t) \geq j$, so the fixed $\tau \in e_0$ otherwise the relative deadline would not be 1 and by the construction of the scheduler $\forall \tau \in e_0. \omega \models_{t+1} \tau$

Theorem 3 (Valid Deadline Scheduler). *Given a well-defined specification φ , an annotation mapping a , and a bound $B = InputEventBound_b$. The scheduler $DS_{\xi_a, B}$ is valid if $\forall \tau \in Tasks. \forall (c, dl) \in \xi_a(\tau). dl > n$ and $b \geq \max_{\tau \in Tasks} |\tau|$ with*

$$n = \min\{n' \mid \forall \tau_1 \tau_2 \dots \in Permutations_{|Tasks|}(Tasks). splits_B(\tau_1 \tau_2 \dots, \varepsilon) = n'\}$$

.

Proof. By definition, we have to proof the following conditions:

1. $\forall \omega \in \llbracket (\varphi, \psi_{\xi_a}, B) \rrbracket. \forall t \in Time. \forall \mathcal{T} \subseteq Tasks. \forall i \in InputValues.$
 $prefix_{DS_{\xi_a, B}}(\omega, t) \wedge DS_{\xi_a, B}(\omega, t) = \mathcal{T} \wedge i \models \mathcal{T}$
 $\rightarrow \exists \omega' \in \llbracket (\varphi, S_a, B) \rrbracket. validTasks(\omega', t+1, \mathcal{T}) \wedge \omega'[..t] = \omega[..t] \wedge \omega'[t+1] = i.$
2. $\forall \omega \in \llbracket (\varphi, \psi_{\xi_a}, B) \rrbracket. \forall t \in Time. \exists t' > t. |DS_{\xi_a, B}(\omega, t')| \geq 1$

We proof each condition separately:

1. We proof this condition by constructing ω' , with $\omega'[..t] = \omega[..t]$, $\omega'[t+1] = i$, and ω' follows the scheduler $DS_{S_a, B}$. Then we have to proof:
 - $\omega \in \llbracket \varphi \rrbracket$ follows from the well-definedness of φ
 - $\omega \in \llbracket \psi_{\xi_a} \rrbracket$ follows from Lemma 2
 - $\forall t \in Time. B(\omega, t)$ and $validTasks(\omega', t+1, \mathcal{T})$ follows from Theorem 2.
2. This condition follows directly from the scheduler $DS_{\xi_a, B}$. By taking the task with the earliest deadlines and the partial order \leq_{DL} over the tasks, the task list is never empty.

Lemma 3. *Given a well-defined specification φ , a world $\omega \in \llbracket \varphi \rrbracket$, an annotation mapping a , and a bound $B = InputEventBound_b$ with $b \geq \max_{\tau \in Tasks} |\tau|$. Then $\omega \in \llbracket \psi_{\xi_a} \rrbracket$ if $\forall t. validTasks(\omega, t, PS_{\xi_a, B}(\omega, t'))$.*

Proof. By unrolling the definition of $\omega \in \llbracket \psi_{\xi_a} \rrbracket$, we get:

$$\forall t \in Time. \forall \tau \in Tasks. \begin{cases} \omega \models_{t+1} \tau & \text{if } \psi_{\xi_a}(\omega, t)(\tau) = Y \\ \top & \text{if } \psi_{\xi_a}(\omega, t)(\tau) = M \\ \omega \not\models_{t+1} \tau & \text{if } \psi_{\xi_a}(\omega, t)(\tau) = N \end{cases}$$

with

$$\psi(\omega, t)(\tau) = \begin{cases} Y & \text{if } (c_1, p_1) = Prio_{\xi}(\tau, \omega, t) \wedge \exists \tau'. (c_2, p_2) = Prio_{\xi}(\tau', \omega, t) \\ & \wedge \omega \models_{t+1} \tau' \wedge p_1 > p_2 \wedge \neg \exists \tau'' \succeq \tau. \omega \models_{t+1} \tau'' \\ M & \text{otherwise} \end{cases}$$

Given an arbitrary but fix $t \in \text{Time}$ and $\tau \in \text{Tasks}$ then, we have to proof:

$$\psi_{\xi_a}(\omega, t)(\tau) = Y \rightarrow t + 1 \models_{\omega} \tau$$

In priority schedules $\psi_{\xi_a}(\omega, t)(\tau) = Y$ is true if there exists a task τ' where the world satisfies this task at the next timestep and this task has a lower priority as τ and τ' is not triggered by a task dependency. Since the world follows PS and PS picks the tasks according to their priority, then the world also satisfies this task τ at the next timestamp or $\tau = \tau_1 \cup \tau_2$ and is for this triggered. However, due to the construction of ξ_a the priority of τ is at least the priority of τ_1 and τ_2 . For this, if τ_1 and τ_2 are in the next taskset so is also τ and the condition holds.

Theorem 4 (Valid Priority Scheduler). *Given a well-defined specification φ , an annotation mapping a , and a bound $B = \text{InputEventBound}_b$. The scheduler $PS_{\xi_a, B}$ is valid $b \geq \max_{\tau \in \text{Tasks}} |\tau|$.*

Proof. By definition, we have to proof the following conditions:

1. $\forall \omega \in \llbracket (\varphi, \psi_{\xi_a}, B) \rrbracket. \forall t \in \text{Time}. \forall \mathcal{T} \subseteq \text{Tasks}. \forall i \in \text{InputValues}.$
 $\text{prefix}_{PS_{\xi_a, B}}(\omega, t) \wedge PS_{\xi_a, B}(\omega, t) = \mathcal{T} \wedge i \models \mathcal{T}$
 $\rightarrow \exists \omega' \in \llbracket (\varphi, \psi_{\xi_a}, B) \rrbracket. \text{validTasks}(\omega', t + 1, \mathcal{T}) \wedge \omega'[..t] = \omega[..t] \wedge \omega'[t + 1] = i.$
2. $\forall \omega \in \llbracket (\varphi, \psi_{\xi_a}, B) \rrbracket. \forall t \in \text{Time}. \exists t' > t. |PS_{\xi_a, B}(\omega, t')| \geq 1$

We proof each condition separately:

1. We proof this condition by constructing ω' , with $\omega'[..t] = \omega[..t]$, $\omega'[t + 1] = i$, and ω' follows the scheduler $PS_{\xi_a, B}$. Then we have to proof:
 - $\omega \in \llbracket \varphi \rrbracket$ follows from the well-definedness of φ
 - $\omega \in \llbracket \psi_{\xi_a} \rrbracket$ follows from Lemma 3
 - $\forall t \in \text{Time}. B(\omega, t)$ and $\text{validTasks}(\omega', t + 1, \mathcal{T})$ follows from Theorem 2.
2. This condition follows directly from the scheduler $PS_{\xi_a, B}$. By taking the task with the earliest deadlines and the partial order \leq_P over the tasks, the task list is never empty.

Theorem 5 (Valid Deadline-Priority Scheduler). *Given a well-defined specification φ , an annotation mapping a , and a bound $B = \text{InputEventBound}_b$. The scheduler $DPS_{\xi_a, B}$ is valid $b \geq \max_{\tau \in \text{Tasks}} |\tau|$.*

Proof. The proof follows the same structure as the proof of Theorem 5 but adds with the *overdue* function an additional priority that is higher than the other priorities. This is reflected in the semantics and in the scheduler