

A Tutorial on Stream-based Monitoring^{*}

Jan Baumeister , Bernd Finkbeiner , Florian Kohn  ,
and Frederik Scheerer 



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



Abstract. Stream-based runtime monitoring frameworks are safety assurance tools that check the runtime behavior of a system against a formal specification. This tutorial provides a hands-on introduction to RTLola, a real-time monitoring toolkit for cyber-physical systems and networks. RTLola processes, evaluates, and aggregates streams of input data, such as sensor readings, and provides a real-time analysis in the form of comprehensive statistics and logical assessments of the system's health. RTLola has been applied successfully in monitoring autonomous systems such as unmanned aircraft. The tutorial guides the reader through the development of a stream-based specification for an autonomous drone observing other flying objects in its flight path. Each tutorial section provides an intuitive introduction, highlighting useful language features and specification patterns, and gives a more in-depth explanation of technical details for the advanced reader. Finally, we discuss how runtime monitors generated from RTLola specifications can be integrated into a variety of systems and discuss different monitoring applications.

Keywords: Monitoring · Specifications · Cyber-Physical Systems

1 Introduction

Runtime monitoring is an applied formal method that assures the safety of a running system by evaluating its behavior against a formal specification. In the stream-based approach, this specification is given in terms of equations that relate input streams, that contain raw data such as sensor readings, to output streams that transform and aggregate the incoming information. The values on the output streams are then checked against trigger conditions that indicate faulty or dangerous situations.

^{*} This work was partially supported by the Aviation Research Program LuFo of the German Federal Ministry for Economic Affairs and Energy as part of "Voloctoper Sicherheitstechnologie zur robusten eVTOL Flugzustandsabsicherung durch formales Monitoring" (No. 20Q1963C), by the German Research Foundation (DFG) as part of TRR 248 (No. 389792660), and by the European Research Council (ERC) Grant HYPER (No. 101055412).

This tutorial provides a comprehensive introduction to the RTLOLA monitoring framework. RTLOLA is the real-time extension [13] of the *Lola* specification language [8], which pioneered the stream-based approach. RTLOLA has been successfully applied to cyber-physical systems such as (unmanned) aircraft. Major case studies include the DLR ARTIS (Autonomous Rotorcraft Testbed for Intelligent Systems) family of aircraft developed at the German Aerospace Center (DLR) [4], and the fully-electric aircraft designed by Volocopter, a leading aircraft manufacturer of electric multi-rotor helicopters [2].

In the tutorial, we develop an RTLOLA specification for a real-world *detect and avoid* problem from the aerospace domain. We consider an autonomous drone flying in a shared airspace. The task of the monitor is to detect aircraft in the vicinity of the drone that might interfere with the drone’s flight path. We will develop the specification in multiple steps, starting with the simple case of a single non-moving object. Our final specification will handle an a priori unbounded number of independently moving entities. Along the way, we introduce the relevant RTLOLA concepts and some fundamental background.

There are two possible ways to read this tutorial. If this is the reader’s first encounter with RTLOLA, we recommend focussing on the development of the *detect and avoid* example. The tutorial starts in Section 2 with an overview of the monitoring framework and the running example. Afterwards, Section 3, Section 4, Section 5 and Section 6 extend the specification step-by-step, each section building up on the previous one. Finally, Section 7 explains how a monitor generated from a specification can be integrated into an existing system.

For readers interested in understanding the RTLOLA approach at a deeper technical level, the tutorial contains subsections with additional background. These subsections are marked as *for experts* to indicate that the subsections can safely be skipped at first reading. In this spirit, the *for experts* subsection of Section 2 provides a comprehensive overview of the various backends available in the RTLOLA framework; in Section 3, we discuss the static analysis of RTLOLA specifications. In Section 4, we introduce the type system, which is further refined in Section 5. In Section 6, we discuss finer points of parameterized specifications.

The tutorial is best experienced when following along in a browser window using the interactive RTLOLA Playground [16], which is available [online](#) [15]. Section 7 briefly explains how this tutorial is integrated into the Playground.

Related Work. There is a rich literature on runtime verification; we refer the reader to several introductory articles (cf. [11,1,19]). A previous tutorial on runtime monitoring has focussed on writing monitors using aspect-oriented programming [10]. Tutorials on stream-based monitoring have appeared as presentations at conferences (cf. [22,20]), but this is, to the best of our knowledge, the first tutorial paper that includes a hands-on development of a stream-based specification for a real-life application scenario. While the paper is based on the RTLOLA framework [13], the fundamental concepts apply in similar form to other stream-based monitoring approaches. Notable other stream-based monitoring approaches, in addition to Lola [8] and its successor Lola2.0 [12], are the Tesla [7] and Striver [18] tools.

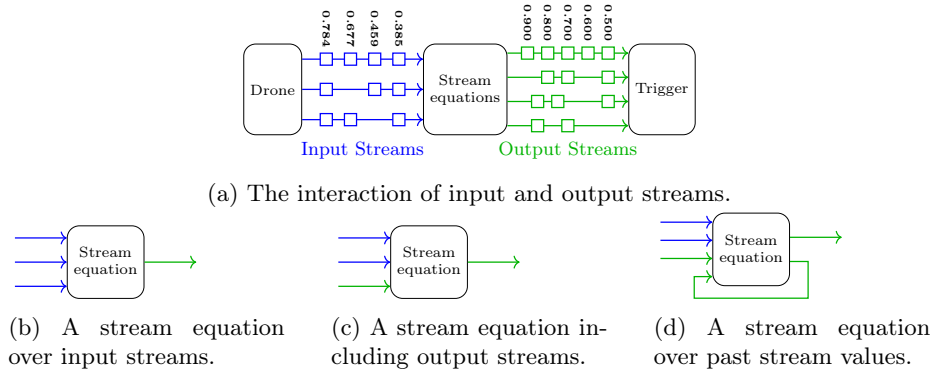


Fig. 1: An overview over stream-based runtime monitoring.

2 Overview

Runtime monitoring checks the behavior of a system, such as cyber-physical system (CPS), at runtime. The monitor receives input data from the system and analyses the data according to a formal specification. At each step, the monitor outputs if the specification is violated such that the system or an operator of the system can initiate countermeasures to return the system to a safe state. The next subsection introduces the general idea of stream-based monitoring followed by an introduction of the running example used in this tutorial.

Stream-based Monitoring

Stream-based runtime monitoring interprets the system’s input data as streams, i.e. a temporal discrete sequence over rich data values, transforms these input streams into output streams and expresses violations in the system’s behavior. Figure 1a illustrates this idea: On the left side of the figure is the monitored system, which is in our case a drone. This system emits a sequence of timed values, which are called input streams. Stream equations then transform these input streams into output streams. Intuitively, stream equations are comparable to variable assignments in imperative programming languages. Yet, instead of only being evaluated once, like variable assignments, they are applied at every stream position to filter incoming data, compare values from different streams, or express complex temporal properties. Stream-based specification languages provide different stream access functions to reason about input streams (see Figure 1b) and other output streams (see Figure 1c). Stream equations can also access past values of a stream (see Figure 1d) to express temporal properties. Finally, special boolean-valued output streams, called trigger streams, express violations in the system’s behavior.

The next section introduces some syntax and semantics of stream equations in more detail, but first, we outline this tutorial’s running example.

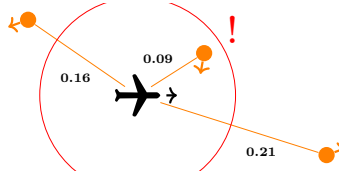


Fig. 2: A drone monitoring surrounding vehicles.

2.1 Running Example

During this tutorial, we build a specification within the aerospace domain inspired by the *Detect and Avoid* specification introduced by Baumeister et.al. [2]. More concretely, we consider an autonomous drone flying in a shared airspace and describe the detection of surrounding vehicles that might interfere with the drone’s flight path. Figure 2 illustrates this property: The plane in the figure’s center represents the drone’s current position and the rays around the plane, the distance to other participants in the airspace. The exclamation mark indicates that this participant is close to the drone and is still approaching, so the drone should change the flight path to avoid a crash. From now on, we will call these participants *intruders* since they might interfere with the drone’s flight path.

Algorithmically, the monitor should perform the following operations to monitor this scenario:

1. *Distance*: The distance to each intruder should be computed whenever the intruder or the drone moves.
2. *Closer*: To check if a specific intruder is approaching the drone, the monitor has to calculate whether the distance of the intruder is decreasing.
3. *Trigger*: If this approach continues over five seconds and the intruder is close to the drone, the monitor should notify the drone to initialize a counteraction.
4. *Stale*: The monitor should check if the intruder positions are regularly updated. Otherwise, it is declared out-of-range.

In this scenario, we expect the drone to provide the monitor with two sensor readings representing the input streams in our specification. First, we assume that the system has a global navigation satellite system (GNSS) to provide the monitor with the drone’s latitude and longitude. Additionally, we assume that the system can detect other vehicles by sharing their position with other participants or by actively searching for intruders, e.g., with a radar, to get the latitude and longitude of the intruders. As output, the monitor provides the current distance to each intruder and the trigger if an approaching intruder is nearby.

The following sections explain different specifications describing these requirements in RTLOLA in a step-by-step fashion. We start with an intruder with a fixed position, e.g., a tree. Then, the specification is adjusted to handle a

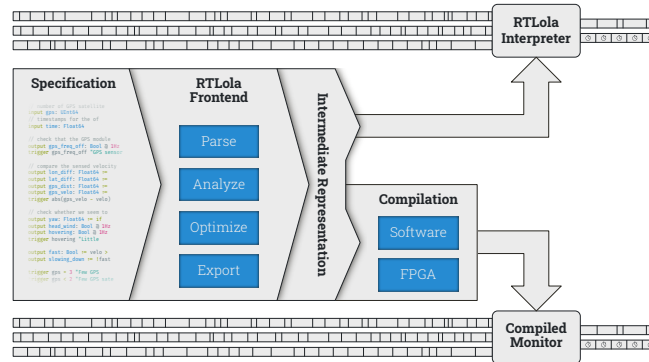


Fig. 3: Overview of the RTLola framework.

moving intruder, e.g., a single moving drone. Finally, we adapt the specification to detect more than one approaching vehicle.

For Experts: Integration & Compilation

The RTLola specification language is embedded in an extensive framework to analyze and monitor specifications. Figure 3 provides an overview of this framework.

It is divided into the frontend and several backends. The frontend takes a specification file and produces an intermediate representation. This representation contains an abstract syntax tree of the specification annotated with additional information relevant to the backends. Additionally, the frontend optimizes the specification as presented in [3]. To verify the functional correctness of the specification, the intermediate representation can be inspected before executing the monitor, as shown in [9].

All backends have online and offline monitoring capabilities, i.e., they can monitor a system at runtime or monitor a log of its execution. RTLola specifications can be executed in the software-based interpreter [13] or compiled into the hardware description language VHDL [5] or imperative programming languages [17]. Providing both an interpretation and compilation ensures flexibility and efficiency: An interpretation allows for easy debugging and quick development times of the specification as it can easily be adjusted and reevaluated. A compilation, particularly a compilation to hardware, can provide highly optimized monitors that meet strict system requirements such as a low power consumption. The framework’s versatility was confirmed in industrial case studies [2] with aerospace partners.

In the RTLola framework, the interpreter takes the specification as its intermediate representation and interprets it based on the incoming data from the system. It provides an extensive API to integrate it into existing implementations. Through the API, the interpreter can quickly adapt to different input data formats and sources.

The compilation takes the intermediate representation and produces executable code that implements a monitor for the given specification. For software, it produces code in an imperative programming language such as Rust. The hardware compiler produces VHDL code that can then be synthesized onto an FPGA. The monitor implementation receives inputs through input wires, and the current stream values are stored on the corresponding output wires or variables. After implementing the communication between the system and the monitor, it can be deployed with the system. Although this approach is less flexible than the interpretation, the resulting monitor is highly efficient once built and integrated.

3 Stream-based Specifications

After Section 2 has introduced the general idea of streams and stream equations, this section presents the concrete syntax of RTLOLA and gives the first concrete examples. Output streams are declared using the `output` keyword followed by a stream expression describing the computation of a stream value. In comparison, input streams are declared using the `input` keyword and do not require a stream expression as their value is given by the system under observation. Trigger streams, special output streams with boolean stream expressions to convey violations to an operator, are defined using the `trigger` keyword.

Stream expressions consist of common arithmetic and logical operators such as addition, subtraction, and conjunction. Higher-level mathematical functions such as sine, cosine, or the square root can be enabled by importing the `math` module. To access past stream values at discrete positions RTLOLA includes the `offset(by: -n)` operator to access the n -th last value of a stream. As the n -th last value of a stream might not exist, as the stream, for example, only has $n-1$ positions yet, an offset operation must be followed by a default value to choose in that case.

Consider the following simplified specification of the scenario explained in Section 2. We limit ourselves to a single non-moving intruder instead of multiple moving intruders and assume a synchronous timing model, i.e., all streams are evaluated when the input streams receive new values.

Example 1 (A Static Intruder).

```

1 | import math
2 | input lat: Float
3 | input lon: Float
4 | constant intruder_lat: Float := 249.301
5 | constant intruder_lon: Float := 23.453
6 |
7 | output distance: Float := sqrt((intruder_lat - lat)**2.0 + (intruder_lon - lon)**2.0)
8 | output closer: Bool := distance.offset(by: -1).defaults(to: distance) >= distance
9 |
10 | trigger closer && distance < 0.1 "Too close to the intruder"

```

The two input streams `lat` and `lon` represent the measurements of the drone's GPS coordinates. The constants below capture the static position of the non-moving intruder. The output stream `distance` keeps track of the Euclidian distance between the drone and the intruder. To achieve this, it retrieves the current values of the input streams and uses the formula to compute the distance. The type of the `distance` stream is automatically inferred and can be omitted, denoted transparently in the example. This inferred type follows from the floating point numbers given by the stream accesses and the underlying functions that operate on this type. The output stream `closer` captures the temporal property that the drone gets closer to the intruder. For that, its stream expression compares the last value of the distance stream, expressed by the `offset`-operator, with the current one. Given that the stream expression consists of a comparison, the type of the `closer` stream is inferred as a boolean automatically. Finally, a trigger stream defines the condition when the distance is too close. To make the trigger more precise, we require that the `closer` stream also evaluates to `true`. Therefore, the trigger only activates if the drone moves towards the intruder.

3.1 Semantics

The semantics of an RTLOLA specification is defined as a relation between input and output streams. Intuitively, it compares every stream value at every time-point with the computed value described by the stream expression. The following definition gives the semantics [8,21] for a subset of RTLOLA to cover the general idea without focusing on concrete details.

Definition 1 (Simplified RTLOLA Semantics). *Let φ be an RTLOLA specification with input stream variables i_1, \dots, i_m and output and trigger stream variables s_1, \dots, s_n . Let τ_1, \dots, τ_m be streams of length N of input values. The tuple $\langle \sigma_1, \dots, \sigma_n \rangle$ of streams of length N is called an evaluation model with respect to τ_1, \dots, τ_m iff for each equation in φ the following holds:*

$$\sigma_i(j) = \text{val}(e_i)(j) \quad \text{for } 0 \leq j \leq N$$

where e_i is the corresponding stream expression of s_i and $\text{val}(e)(j)$ is for the expressions in our example defined as:

$$\begin{aligned} \text{val}(c)(j) &= c \\ \text{val}(i_t)(j) &= \tau_t(j) \\ \text{val}(s_t)(j) &= \sigma_t(j) \\ \text{val}(f(e_1, \dots, e_k))(j) &= f(\text{val}(e_1)(j), \dots, \text{val}(e_k)(j)) \\ \text{val}(e.\text{offset}(\text{by: } i).\text{defaults}(\text{to: } d))(j) &= \begin{cases} \text{val}(e)(j+i) & \text{for } 0 \leq j+i \\ \text{val}(d)(j) & \text{otherwise} \end{cases} \end{aligned}$$

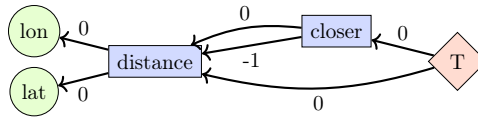
3.2 Evaluation Algorithm

In contrast to imperative programs, the order of the equations in a stream-based specification does not influence the order of the evaluation. They are generally

evaluated simultaneously, yet accesses between streams imply dependencies between streams and therefore affect the order. We represent these dependencies in a graph-based representation called the dependency graph. An analysis of this graph then computes a correct order of the stream evaluation: Every topological order of the dependency graph represents a valid order to process the streams during a single evaluation cycle.

Definition 2 (Dependency Graph). *Let ϕ be an RTLOLA specification. The dependency graph of ϕ is a directed weighted multi-graph $G = \langle V, E \rangle$ with $V = \{i_1, \dots, i_m, s_1, \dots, s_n\}$. An edge $e = \langle s_i, s_k, w \rangle$ is in E iff the expression of s_i contains $s_k.offset(by: w)$ as a sub-expression. Analogously, edges with weight 0 are added for non-offset accesses.*

Example 2 (Dependency Graph). The following graph describes the dependency graph for the specification in Example 1:



Every node in the graph corresponds to a stream in the specification. For a better illustration, we mark input streams green, output streams blue, and trigger streams red. The input streams `lat` and `lon` do not have outgoing edges since input streams represent the input data and do not have a stream expression. The output stream `distance` accesses the current value of the `lon` and `lat` stream in the computation resulting in the 0-edge in the dependency graph. The `closer` stream accesses the current and last value of the `distance`-stream, resulting in a zero and an offset edge. Unlike input streams, trigger streams have no incoming edges since no stream expression can access these streams. In our example, the outgoing edges of the trigger are the accesses to the `distance` and `closer` stream.

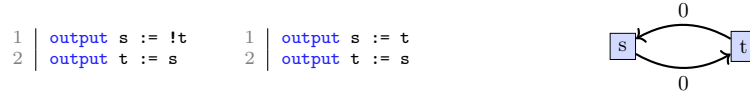
Using this graph, we can compute different evaluation orders for this specification ensuring that the monitor accesses the intended stream values: The specification allows every order in which the `distance`-stream is evaluated after the inputs, the `closer`-stream after the `distance`-stream, and at the end of the evaluation the trigger stream.

For Experts: Static Analysis

This expert subsection presents two static analyses based on the dependency graphs that guarantee a safe evaluation of stream-based specifications. The first analysis guarantees the existence of a unique evaluation model, whereas the second analysis computes an upper bound of the required memory. The latter analysis can determine if the monitor can run in a resource-constrained environment before execution.

Well-formed Specifications With this analysis, we define a syntactic criterion to guarantee the existence of a unique evaluation model. In general, we cannot find a unique evaluation model, if we cannot determine a order of the stream evaluation. This problem corresponds to a cycle in the dependency graph. First, consider the following examples of syntactically valid specifications and their dependency graph illustrating the problem.

Example 3 (Ill-defined Specifications and their Dependency Graph).



According to the semantics, the specification on the left has no evaluation model, since we cannot find an assignment for s and t that satisfies both equations. Such specifications are not well-defined and should be rejected by the RTLOLA framework. In comparison, the specification on the right has multiple evaluation models as long as s and t are equal. However, for this specification, we again cannot compute a valid evaluation order. As a result, neither specification can be evaluated algorithmically.

Both specifications have the same graph with an edge from stream s to stream t and an edge in the other direction, resulting in a cycle. Because of this cycle, we cannot determine a valid evaluation order and should reject the specification. We can give a syntactic criterion for well-definedness called well-formedness based on the dependency graph of a specification [8]:

Definition 3 (Well-formedness). *A specification is well-formed, iff for every cycle in its dependency graph, the accumulated edge weight of the cycle is not zero.*

As described in the definition, we do not forbid every cyclic behavior. The next example shows a valid specification containing a cycle in the dependency graph. It sums all values of the input stream a :

Example 4 (Valid Cycle).

```
1 | input a : Int
2 | output sum := sum.offset(by: -1).defaults(to: 0) + a
```

Here, the `sum` uses the offset expression to access the past value of itself. This access results in a cycle in the dependency graph, but the sum of this cycle is negative. Intuitively, this is allowed since past values are already computed, and we can ignore these accesses when building the evaluation order.

Static Memory Bounds Secondly, we can determine the amount of values that need to be kept in memory for every stream. Again, this analysis is based on the dependency graph and allows giving static memory bounds for specifications.

Definition 4 (Memory Bound). Let $G = \langle V, E \rangle$ be the dependency graph of the specification φ . For every stream s in φ its memory bound is determined as $\max(\{-w \mid \langle o', s, w \rangle \in E\})$.

Intuitively, the memory-bound of a stream is defined by the largest offset at which the stream is accessed. All values before that bound are not required for further computation and can be discarded. For the specification in Example 1 the memory bound of all streams is zero except for the `distance` stream. Given that the stream is accessed with an offset of -1 , the memory bound of this stream is one. The memory bound of the specification as a whole is then the sum over the memory bounds of all input and output streams.

4 Event-based & Periodic Streams

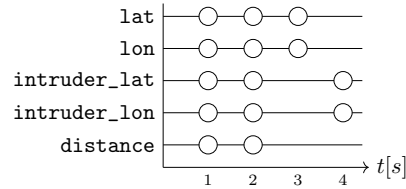
This section lifts the simplification of a static intruder to a moving intruder. For this, consider the following naïve extension of Example 1, where the intruder position is given as additional input streams `intruder_lat` and `intruder_lon`.

Example 5 (A Synchronous Moving Intruder).

```

1 | import math
2 | input lat: Float
3 | input lon: Float
4 | input intruder_lat: Float
5 | input intruder_lon: Float
6 |
7 | output distance :=
8 |   sqrt((intruder_lat - lat)**2.0
9 |     + (intruder_lon - lon)**2.0)
10 | output closer := distance.offset(by: -1)
11 |   .defaults(to: distance) >= distance
12 |
13 | trigger closer && distance < 0.1
14 |   "Too close to the intruder"

```



This specification is correct in a synchronous setting, i.e. a setting where all input streams receive a new value simultaneously. Yet, in reality, the intruder and the drone are independent systems. Hence, the measurements of the intruder's position might not be synchronous with the drone's position measurements. In an asynchronous setting, input streams receive values independent of each other. For this, every output stream and trigger is only evaluated if the input streams they (transitively) depend on receive a new value at the same time. Consider the specification from Example 5 in a synchronous and asynchronous setting as illustrated by the trace on the right. First, all values are received synchronously and the `distance` stream is computed. It might seem correct, yet all output streams and trigger streams still transitively or directly depend on all input streams. As a result, the specification is effectively synchronous again and behaves invalidly if not all inputs receive a new value at the same time, as depicted later in the trace. As we can see, if the current position and the intruder position are not synchronized, the `distance` stream is not updated as expected.

The next example depicts the corrected specification, where input streams are accessed asynchronously using `hold`-accesses:

Example 6 (A Moving Intruder).

```

1 | import math
2 | input lat: Float
3 | input lon: Float
4 | input intruder_lat: Float
5 | input intruder_lon: Float
6 |
7 | output distance @((intruder_lat && intruder_lon) || (lat && lon)) :=
   |   sqrt((intruder_lat.hold(or: 0.0) - lat.hold(or: 0.0))**2.0 +
   |     (intruder_lon.hold(or: 0.0) - lon.hold(or: 0.0))**2.0)
8 | output closer @((intruder_lat && intruder_lon) || (lat && lon)) :=
   |   distance.offset(by: -1).defaults(to: distance) >= distance
9 |
10 |
11 | trigger @1Hz closer.aggregate(over_exactly: 5s, using: forall).defaults(to: false) &&
   |   distance.hold(or: 1.0) < 0.1 "Too close to the intruder"

```

While the `distance` stream mathematically performs the same computation, a `hold()` lookup is used to avoid a direct dependency. This lookup refers to the most current available value of the accessed stream and does not require that the accessed value is computed at the same time. However, there might not be such a value when the accessing stream is evaluated, so a default value must be supplied similar to the offset operator. As the timing of the `distance` stream is decoupled from any input stream, it has to be explicitly specified when it should produce new values. Syntactically, this is specified through a positive boolean expression over input streams following the `@` after a stream's name. This expression is called the *activation condition* of the stream and symbolically describes the events at which the stream is evaluated. In the example, the `distance` stream is evaluated whenever the intruders *or* the drone's position changes. For the `closer` stream the activation condition is the same as for the `distance`, but is automatically inferred because of the synchronous access.

Moreover, the trigger in the specification has changed. It is now periodic, a concept which will be introduced in the subsequent section.

4.1 Periodic Streams

In reality, it is often required that a monitor not only reacts to system actions but can also proactively produce verdicts about the system's health. Otherwise, the monitor could not detect a frozen system because it would freeze as well. In RTLOLA, proactive monitoring is achieved through streams evaluated at a fixed frequency called periodic streams. A periodic stream can be specified by giving a frequency or period annotation like `1Hz` or `1s` after the `@` keyword. These frequencies are independent of input streams and to access input streams from periodic output streams we can either use `hold`-accesses or sliding windows. In our example, we use a sliding window in the trigger stream to make the specification more robust against GPS fluctuations.

Sliding window aggregations aggregate over every value in a given time frame of a stream using an aggregation function. More concretely, the window in our

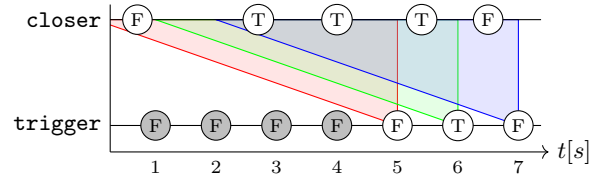


Fig. 4: The functioning of sliding windows exemplified based on Example 6

example evaluates only to true if every `closer` value in the last 5 seconds is true. The functionality of this sliding window is visualized in Figure 4. In our specification, the trigger stream is evaluated with a fixed frequency of one second whereas the `closer` stream depends on the inputs. For the first four seconds, the window does not aggregate any values given that the whole duration is not available at the start of the monitoring. In this case, the window evaluates to the default value `false` instead. Afterwards, the window aggregates over different numbers of `closer` values as illustrated by the different colors.

RTLOLA supports a set of aggregation functions as `exists`, `avg`, `sum`, `count`, `min` or `max`.

For Experts: RTLOLA's Type System

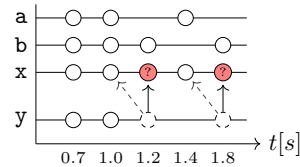
Similar to many programming languages, RTLOLA has a type system that ensures only valid specifications can be executed. The type-system of RTLOLA is twofold: The value type of a stream describes how the memory should be interpreted, i.e., as a signed or unsigned integer, a floating point number, or a string. The underlying value type system ensures that only compatible values are used in an operation, such that, for example, no string and number are added. The pacing type of a stream describes its timing behavior, i.e., it determines the points in time when the stream is evaluated. This pacing type system ensures that every direct access, also called synchronous access, is valid. Consider the following example specification with an invalid synchronous access:

Example 7 (Invalid Synchronous Access).

```

1 | input a: Bool
2 | input b: Bool
3 | output x @a := a
4 | output y @b := x

```



The specification on the left has two output streams: The stream `x`, evaluated whenever the input `a` receives a new value, with that same value of `a`. The stream `y` is evaluated whenever the input `b` receives a new value and takes the value of `x`. The diagram on the right exemplifies the timing behavior of the streams. At time 1.2 and 1.8, only stream `b` receives a new value. Conse-

quently, the stream x is not computed, and with that stream y cannot access the value of x at these points in time. These timing errors can lead to invalid memory accesses at runtime, so the type checker rejects the specification. To fix the specification, one could use a `hold` access from y to x similar to Example 6, resulting in the greyed-out arrows in the diagram on the right.

The Pacing Type System Intuitively, the pacing type system declares a synchronous access valid if the accessed stream is evaluated at least at the same points in time as the accessing stream. For non-periodic, also called event-based streams, this property can be checked by asserting logical implication between the activation condition of the accessing stream and the accessed stream. In Example 7, it is easy to see that $a \rightarrow b$ does not always hold, but, for example, $a \wedge b \rightarrow a$ does. A similar condition holds for periodic streams: A synchronous access between periodic streams is valid if the accessing stream runs at a slower pace than the accessed stream. Concretely, the period of the accessed stream has to be a multiple of the period of the accessing stream. In addition, synchronous accesses between periodic and event-based streams are never valid.

5 Stream Lifecycle

In this section, we optimize the specification from Example 6 by guarding computationally heavy operations with cheap-to-evaluate predicates. Concretely, we only compute the `distance` and `closer` streams and the trigger when the intruder is in range. These optimizations are enabled by lifting the assumption that a stream exists from the start to the end of the monitor. Instead, we allow for dynamic stream creation, i.e., a stream can be created and removed from the monitor at runtime. In the following, we refer to the creation of streams as their `spawn` behavior and their deletion as their `close` behavior. In between its `spawn` and `close` action, a stream is evaluated as defined by its stream expression.

Syntactically, the three steps of a stream’s lifecycle are specified by three sub-clauses in the stream’s definition:

```

1 | output o
2 |   spawn @p_s when c_s
3 |   eval @p_e when c_e with e
4 |   close @p_c when c_c

```

Each of these clauses can feature a pacing and a boolean stream expression preceded by the `when` keyword. The pacing of a clause *statically* determines when the clause could be evaluated, i.e., whenever event `a&b` occurs or at a frequency of `1Hz`. The stream expression preceded by `when` is evaluated at the time points described by the pacing. It constrains these time points dynamically based on runtime values, i.e., the action corresponding to the clause is only taken *when* this expression evaluates to true. The evaluation clause features a stream expression defining *how* the stream’s value is computed after the `with` keyword.

As for the previous sections, pacings can be omitted and inferred by the type checker for brevity. The `when` expression of a clause can also be omitted, representing the constant `true`. If a stream's `spawn` or `close` clause is omitted, the stream exists from the start or until the end of the monitor, respectively. If only an `eval with` clause exists, we can use the short-hand notation `:=` as used in the previous sections.

Consider the following extension of Example 6 in which every stream in the specification is now composed of three clauses: a `spawn` clause, an `eval` clause, and a `close` clause.

Example 8 (An Out-of-Range Intruder).

```

1  import math
2  input intruder_lat: Float
3  input intruder_lon: Float
4  input lat: Float
5  input lon: Float
6
7  output distance
8    spawn @(intruder_lat && intruder_lon)
9    eval @((intruder_lat && intruder_lon) || (lat && lon))
10   with sqrt((intruder_lat.hold(or: 0.0) - lat.hold(or: 0.0))**2.0 +
11            (intruder_lon.hold(or: 0.0) - lon.hold(or: 0.0))**2.0)
12   close @true when stale.hold(or: false)
13 output closer
14   spawn @(intruder_lat && intruder_lon)
15   eval with distance.offset(by: -1).defaults(to: distance) >= distance
16   close @true when stale.hold(or: false)
17 output stale
18   spawn @(intruder_lat && intruder_lon)
19   eval @10s with intruder_lat.aggregate(over: 10s, using: count) = 0
20   close @true when stale.hold(or: false)
21 trigger
22   spawn @((intruder_lat && intruder_lon) || (lat && lon))
23   when distance.hold(or: 1.0) < 0.1
24   eval @1Hz when closer.aggregate(over_exactly: 5s, using: forall).defaults(to: false)
25   with "Intruder detected"
26   close @true when stale.hold(or: false)

```

Another notable change is the added `stale` stream. It defines when the intruder is considered out of range by checking for any GPS coordinate updates in the last 10 seconds. The `spawn` clause of the stream defines that this condition is only monitored once a GPS location is received from the intruder. The omitted `when` expression in the `spawn` clause is equivalent to a `when true` definition. The `close` clause of the stream defines that the condition should no longer be monitored as soon as it becomes true for the first time. However, the stream is spawned again if a new intruder GPS location reactivates the `spawn` condition of the stream.

The `distance` and `closer` streams inherit the same `spawn` and `close` clauses as the `stale` stream, further excluding redundant computations. The trigger also inherits the same close condition as the other two streams. Yet, its `spawn` condition differs slightly. The `distance.hold(or: 1.0) < 0.1` expression is moved from the main trigger condition to its `spawn` condition, adapting the `spawn` activation condition accordingly. That way, the computationally heavy sliding window aggregation is only performed once the intruder is close enough.

Besides optimizing specifications, dynamic stream creations increase the expressiveness of the RTLOLA specification language as shown in the next subsection.

5.1 Deadline Watchdogs

Deadline watchdogs are common specification requirements in CPS and can be expressed in natural language as follows: "*t* seconds after event *e* a condition *c* must hold." Such requirements can be represented in RTLOLA using dynamically created streams:

Example 9 (A Deadline Watchdog).

```

1 | output timer
2 |   spawn when e
3 |   eval @ts with true
4 |   close when timer
5 |
6 | trigger
7 |   spawn when e
8 |   eval @ts when !c with "The deadline was missed"
9 |   close when timer
    
```

For the watchdog, we define a new stream `timer` that is spawned with the start of the watchdog, i.e. the event *e* spawns the watchdog. This is the starting point of the annotated frequency, so this stream is evaluated for the first time *t* seconds after *e*. Since this value is immediately true, we also close the stream after these *t* seconds. The trigger stream has the same `spawn` and `close` condition and is evaluated with the same frequency. Here, we check if the condition *c* is satisfied and use the `timer` stream as a helper function to immediately close the trigger stream after the first evaluation.

For Experts: Semantic Types

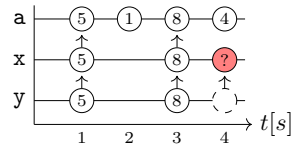
The introduced `when`-conditions further refine the timing of a stream and add another point of failure for synchronous accesses. This problem is solved with another type system.

Semantic Types and Event-based Streams The following example illustrates a possible point of failure using synchronous accesses and `when`-conditions:

Example 10 (Invalid Semantic Types).

```

1 | input a: Int
2 | output x
3 |   eval @a when a > 4 with a
4 | output y
5 |   eval @a when a > 3 with x
    
```



As specified by the `when` expression of the `eval` clause of `x`, it only produces a value when the input `a` is greater than 4. This might not coincide with `a > 3` as required for the evaluation of `y`. Therefore, similar to Example 7, there are points in time when `y` is evaluated, but `x` is not. To detect specifications

with such errors, RTLOLA uses another type system reasoning about the when conditions, called semantic types in that context. Like the pacing type system, the semantic type system ensures the implication relation between semantic types holds. In the above example, the type system would reason whether the implication $a > 3 \rightarrow a > 4$ is a tautology and consequently reject this specification.

Besides ensuring that the *when* conditions of the eval clauses of dependent streams imply each other, the semantic type system also reasons about the lifecycle of dependent streams. Concretely, for a synchronous access to succeed, the accessed stream must be alive at least as long as the accessing stream.

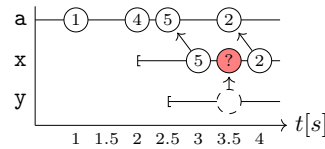
Semantic Types and Periodic Streams While the previous intuition holds for event-based streams, synchronous accesses between periodic streams imply stricter requirements. Consider the following example:

Example 11 (Shifted Periodicity).

```

1 | input a: Int
2 | output x
3 |   spawn when a > 3
4 |   eval @1Hz with a.hold(or: 0)
5 | output y
6 |   spawn when a > 4
7 |   eval @1Hz with x

```



Here the streams x and y have the same frequency and the spawn condition of y implies the spawn condition of x . However, the synchronous access might fail if x spawns before y since the frequencies are now out of sync. For example, the sequence of events depicted on the right of the example will lead to an invalid synchronous access. First, a has the value 1 and since no spawn condition is true both streams are not spawned. Next, a gets the value 4 spawning x but not y . The later stream is spawned half a second later, which results in the two periods of x and y not being synchronized. This leads to the failure of the synchronous access. To circumvent this problem, the semantic type system requires equality instead of the implication of the spawn and close condition of two dependent periodic streams.

Type System Decidability Pacing types are defined as positive boolean formulas over input stream names or as fixed frequencies. Hence, it is efficiently decidable if their implication is a tautology. On the contrary, semantic types are arbitrary stream expressions. As described by Schwenger [21], whether an implication between stream expressions is a tautology is generally undecidable. Nevertheless, the RTLOLA frontend provides a sound over-approximating implementation of the semantic type checker based on syntactic equality.

Here, semantic types are parsed as a boolean formula, so implications such as $a.\text{hold}(\text{or}: 0) \wedge b \rightarrow a.\text{hold}(\text{or}: 0)$ can still be proven.

6 Parameterization

The specifications in the previous sections were limited to a single intruder. However, in reality, the monitor must observe an unbounded amount of intruders since we can not give an a priori bound on their number. This monitor will inevitably require unbounded memory, yet keeping the memory footprint of the monitor predictable is essential for CPS. For this, RTLOLA features parameterized output streams [12] that provide a declarative and predictable way of handling unbounded memory through stream expressions.

Parameterized streams lift output streams from a single instance to a set of stream instances. While all stream instances share the stream expressions, each instance of a stream has a different assignment of parameter values. This assignment is determined by an additional stream expression preceded by the `with` keyword in the `spawn` clause. If an output stream is parameterized over multiple parameters, this expression returns a tuple of values matched position-wise to the parameters. Parameters are declared as a comma-separated sequence in braces after the stream name. The `spawn` clause of a parameterized stream determines when an instance is created as introduced in Section 5. The evaluation and close clauses of parameterized streams are evaluated for each instance, determining their value and lifecycle. As for dynamic streams, a stream instance will not be spawned again if an instance with the parameter values already exists.

Consider the final iteration of the running example which extends each output stream with a parameter for the different intruders:

Example 12 (Multiple Intruder).

```

1 | import math
2 | input lat: Float
3 | input lon: Float
4 | input intruder_id: UInt
5 | input intruder_lat: Float
6 | input intruder_lon: Float
7 |
8 | output intruder_pos(id)
9 |   spawn with intruder_id
10 |     eval when id = intruder_id with (intruder_lat, intruder_lon)
11 |     close @true when stale(id).hold(or: false)
12 | output distance(id)
13 |   spawn with intruder_id
14 |     eval @((intruder_id && intruder_lat && intruder_lon) || (lat && lon))
15 |     with sqrt((intruder_pos(id).hold().0.defaults(to: 0.0) - lat.hold(or: 0.0))*2.0 +
16 |               (intruder_pos(id).hold().1.defaults(to: 0.0) - lon.hold(or: 0.0))*2.0)
17 |     close @true when stale(id).hold(or: false)
18 | output closer(id)
19 |   spawn with intruder_id
20 |     eval with distance(id).offset(by: -1).defaults(to: distance(id)) >= distance(id)
21 |     close @true when stale(id).hold(or: false)
22 | output stale(id)
23 |   spawn with intruder_id
24 |     eval @10s with intruder_pos(id).aggregate(over: 10s, using: count) = 0
25 |     close @true when stale(id).hold(or: false)
26 | trigger(id)
27 |   spawn when distance(intruder_id).hold(or: 1.0) < 0.1 with intruder_id
28 |     eval @1Hz when closer(id).aggregate(over_exactly: 5s, using: forall).defaults(to:
29 |       false) with "Intruder {} detected".format(id)
30 |     close @true when stale(id).hold(or: false)

```

Notice the additional input stream `intruder_id`. We assume that every intruder has a unique ID provided to the monitor with every update of the `intruder_lat` and `intruder_lon` streams. The output stream `intruder_pos` is added to accumulate these positions on a per-intruder basis. It has a single parameter representing the intruder ID. This is made explicit through its `spawn with` expression that synchronously accesses the `intruder_id` stream. As a result, a new instance of this stream is created for each fresh intruder when it is received for the first time. All other output streams share the same parameterization, so each output stream has an instance for each non-stale intruder ID. The trigger features the additional `spawn when` condition as before. Finally, this specification achieves the goal outlined in Section 2 and handles multiple moving intruders efficiently and predictably.

For Experts: Instance Aggregations

Like sliding window aggregations, RTLOLA features instance aggregations to aggregate the most recent values of the instances of a parameterized output stream. For example, the trigger from Example 12 can be rewritten without parameters using an instance aggregation as follows:

```

1  output approaching(id)
2     spawn when distance(intruder_id).hold(or: 1.0) < 0.1 with intruder_id
3     eval @1Hz with closer(id).aggregate(over_exactly: 5s, using:
4         forall).defaults(to: false)
5         close @true when stale(id).hold(or: false)
6  trigger @true approaching.aggregate(over_instances: all, using: exists) "Intruder
   detected"
```

The `approaching` stream is similar to the trigger from Example 12. The non-parameterized trigger now aggregates over all instances of this new stream using a disjunction. Semantically, this means that whenever any instance of the helper stream evaluates to true, the instance aggregation in the trigger will also evaluate to true, causing the trigger to activate.

Instead of aggregating all stream instances, it is sometimes desirable to only aggregate the instances that produced a *fresh* value in this evaluation cycle. This is specified by stating `over_instances: fresh` in the aggregation. This will couple the pacing of the caller of the aggregation to the evaluation pacing of the target stream of the aggregation. Other instance aggregation functions in RTLOLA are `forall`, `avg`, `sum`, `count`, `min`, and `max`.

7 Development and Integration

During the previous sections, the examples featured links to the RTLOLA playground, a web-based implementation of the RTLOLA frontend and interpreter that can analyse and execute specifications locally in your browser without requiring installation. A version of this tutorial is also available there¹, such that the specifications and examples from this tutorial can easily be experimented

¹ <https://rtlola.org/playground/tutorial>

with by pressing the *Copy to Editor* button below them and clicking *Run*. The specifications are tested against a trace simulated using the *Microsoft Flight Simulator*. A video of this trace is included in the overview chapter in the playground, and its raw data can be inspected by switching to the *Trace* tab on the right.

Nonetheless, in real applications, it is necessary to run the monitor either natively or incorporate them within existing applications. The RTLOLA interpreter provides solutions for both: A library to seamlessly integrate the monitor into Rust applications, along with a standalone command-line application.

7.1 RTLOLA CLI

The simplest way to run the RTLOLA interpreter locally is the command-line application `rtlola-cli`. The installation of the application can be achieved using the cargo package manager:

```
1 | cargo install rtlola-cli
```

The application offers two modes of execution:

- *Analyze* In this mode, the RTLOLA frontend is employed to check the provided specification for correctness. This involves checking for syntax errors, ensuring well-definedness and detecting type-related errors.
- *Monitor* This mode enables the execution of a monitor based on the provided specification. After checking the specification for correctness, the monitor can be run in an offline or online setting. In an offline setting, the monitor analyzes a prerecorded trace, while in an online setting, the data arrives in real-time.

In the offline setting, the interpreter monitors a prerecorded trace in CSV format such as the following:

```
1 | a,b,time
2 | 1,2,0
3 | #,3,1
4 | 3,#,2
```

This example trace defines three events occurring at times 0s, 1s, and 2s, respectively, and assigns new values to two input streams called `a` and `b`. Notably, the hashmark `#` signifies that the corresponding input stream does not receive a new value at that particular time. Subsequently, we can run the `rtlola-cli` to monitor the specification `specification.lola` on this trace:

```
1 | rtlola-cli monitor \
2 |   --offline relative \
3 |   --csv-in trace.csv \
4 |   specification.lola
```

Here, the argument `--offline relative` specifies the type of time format utilized in the "time" column of the CSV file. In the example, the timestamp is given as time in seconds relative to the start of the monitor. Upon executing, each event in the CSV file is forwarded to the monitor, with the resulting output printed to the standard output.



Fig. 5: Process to integrate the monitor

In the online mode, the interpreter retrieves the inputs from a buffer and utilizes the real-time timestamps of the events when they arrive. The following command starts the monitor in an online setting:

```
1 | rtlola-cli monitor --online --stdin specification.lola
```

Here, the application waits for new events on standard input and forwards them to the monitor as soon as they arrive.

We illustrated two instances of using the RTLOLA interpreter command line application. For a comprehensive list of available command-line arguments and options, you can consult the documentation by executing the following command:

```
1 | rtlola-cli monitor --help
```

The RTLOLA interpreter has successfully been employed to monitor drones in cooperation with the German Aerospace Center (DLR) and the aircraft manufacturer Volocopter. We identified a set of different monitoring applications, as reported by Baumeister et.al. [2], which we discuss in the expert section.

For Experts: Monitoring Applications

The monitor can provide valuable feedback during the development of new components in the aerospace domain. In this safety-critical domain, predefined standards ensure that the concept of operation, requirements, design, and implementation are coherent and include several validation steps. We identified different applications in which monitoring can be used during the development of new components following such standards but also during the operation of these components:

1. *Debugging* The monitor provides feedback to the developer of the component. During the execution, the monitor checks whether the component works as intended and collects statistical information. The developer writes the specification and has access to the internal state of the component.
2. *Validation* The monitor is used to validate the component externally. The specification is written independently and has only access to the inputs and outputs of the component and not to the internal state. This application is, among others, helpful in validating that components by external companies follow their requirements and can be trusted.
3. *Pre-Post-Flight Analysis* This application uses monitoring to check whether all necessary components are operational. The monitor runs predefined test cases and validates that no irregular behavior is detected. Af-

ter the flight, the monitor computes more sophisticated information for better evaluation of the flight or runs new specifications on past flights.

4. *In-Flight Analysis / Safe Integration* The monitor provides feedback about the safety of the drone during its operation. It validates the correctness of individual components to ensure a safe flight and reports to the pilot if a property is violated.

All these applications require the integration of the monitor into the development process. Previously, we utilized the command-line application of the RTLOLA interpreter to execute the monitor. In the following example, we demonstrate how to integrate the interpreter using the Rust library.

Integration with the RTLola API We assume that the monitor is running on the drone and receives the input data over internal communication. The output of the monitor is then sent over TCP to a ground station displaying the trigger messages of the monitor so a pilot can take over. Figure 5 illustrates two steps for this integration process. As shown in the figure, the setup consists of three components: The monitor, in the center of the figure, is automatically generated from the specification and does not require further integration. However, two interfaces must be implemented to handle the communication with the system and the operator. These implementations are specific to the setup: The *Event Conversion* receives the incoming sensor readings and transforms the data into an internal representation the monitor understands. The *Verdict Conversion* transforms the internal representation of the monitor’s output to messages accepted by the ground station.

After providing all the missing implementations, we need to configure the monitor and start the evaluation. This results in the following code, skipping the concrete configuration of the `event_source` and the `verdict_sink`:

```

1 | let mut monitor = ConfigBuilder::new()
2 |   .with_spec(spec)
3 |   .online()
4 |   .with_event_factory::<ExampleInputs::Factory>()
5 |   .with_verdict::<TriggerMessages>()
6 |   .monitor()?;
7 | let mut event_source = ...
8 | let mut verdict_sink = ...
9 | while let Some((ev, ts)) = event_source.next_event()? {
10 |     let verdicts = monitor.accept_event(ev, ts)?;
11 |     verdicts_sink.sink_verdicts(&verdicts)?;
12 | }
```

Event Conversion The Event Conversion receives sensor values and transforms the readings into an internal representation the monitor uses. In our setup, we receive the sensor values over a UDP connection as a byte-stream and differentiate between two types of messages: The first type of message is sent by the GNSS sensor that computes the latitude and longitude of the drone. The second type of message contains the latitude and longitude of the intruders. The byte-stream needs to be parsed and converted to map the incoming data to the corresponding input streams. The implementation in our

```

1 | impl ByteParser for DroneExampleParser {
2 |     fn from_bytes(&mut self, bytes: &[u8]) -> Result<(ExampleInputs, usize)> { ... }
3 | }

1 |                                     1 | #[derive(ValueFactory)]
2 |                                     2 | #[factory(prefix)]
1 | #[derive(ValueFactory)] 3 | struct Intruder {                1 | #[derive(CompositFactory)]
2 | struct Gnss {              4 |     id: UInt64                    2 | enum ExampleInputs {
3 |     lat: Float64,         5 |     lat: Float64,                3 |     Gnss(Gnss),
4 |     lon: Float64         6 |     lon: Float64                4 |     Intruder(Intruder)
5 | }                          7 | }                                5 | }

```

Fig. 6: Concrete Implementation of the Event Conversion

setup uses the simplified interfaces shown in Figure 6. After providing a parser function, the implementation receives the byte stream and parses this stream to an event called `ExampleInputs`. The implementation of the interface is provided automatically by the macros `ValueFactory` and `CompositFactory`. These macros generate code for a factory that maps, for example, the `lat` field in the `Gnss` struct to the `lat` input stream in the monitor.

Verdict Conversion The Verdict Conversion transforms the internal representation of the monitor’s output into messages in a form expected by the ground station. In our setup, this conversion interprets trigger messages as bytes and sends these over TCP to the ground station. The interfaces for this setup are implemented generically, so no further steps are needed.

8 Conclusion

The running example from our tutorial has illustrated the expressiveness of the RTLOLA specification language, which makes RTLOLA well-suited for complex application domains like aerospace. The development of the specifications is facilitated by the comprehensive support of the tool framework. Since the same specification can be used in multiple different settings, a specification can be validated early in a test environment or on log data, long before the monitor is integrated into the aircraft; the automatic analysis of the specification furthermore ensures that the monitor operates correctly and reliably.

RTLOLA has been very successful in the aerospace domain (cf. [4,2]). RTLOLA has also been used in other cyber-physical applications, including cars [6] and medical equipment [14], and in domains beyond CPS, such as networks [12]. The combination of the highly expressive RTLOLA specification language with the reliability obtained by static analysis and the resource efficiency of the monitoring framework is of great use in all these settings.

Data Availability Statement The artifacts and resources associated with this paper are accessible as follows:

1. **Primary Artifact:** The primary artifact of this paper is available via:
<https://doi.org/10.5281/zenodo.12633784>.
2. **Source Code:** The source code of the framework is hosted on GitHub:
 RTLola Interpreter: <https://github.com/reactive-systems/RTLola-Interpreter>
 RTLola Frontend: <https://github.com/reactive-systems/RTLola-Frontend>
3. **Software Packages:** The relevant software packages are available on crates.io:
 RTLola Interpreter: <https://crates.io/crates/rtlola-interpreter>
 RTLola Frontend: <https://crates.io/crates/rtlola-frontend>

References

1. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) *Lectures on Runtime Verification - Introductory and Advanced Topics*, Lecture Notes in Computer Science, vol. 10457, pp. 1–33. Springer (2018). https://doi.org/10.1007/978-3-319-75632-5_1
2. Baumeister, J., Finkbeiner, B., Kohn, F., Schirmer, S., Torens, C., Löhr, F., Manfredi, G.: Monitoring unmanned aircraft: Specification, integration, and lessons-learned. In: *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, Canada, July 22-27, 2024*. Accepted for publication (2024)
3. Baumeister, J., Finkbeiner, B., Kruse, M., Schwenger, M.: Automatic optimizations for stream-based monitoring languages. In: Deshmukh, J., Nickovic, D. (eds.) *Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings*. Lecture Notes in Computer Science, vol. 12399, pp. 451–461. Springer (2020). https://doi.org/10.1007/978-3-030-60508-7_25
4. 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
5. 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>
6. Biewer, S., Finkbeiner, B., Hermanns, H., Köhl, M.A., Schnitzer, Y., Schwenger, M.: On the road with rtlola. *Int. J. Softw. Tools Technol. Transf.* **25**(2), 205–218 (2023). <https://doi.org/10.1007/S10009-022-00689-5>
7. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: Tessler: 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*. Lecture Notes in Computer Science, vol. 11254, pp. 144–162. Springer (2018). https://doi.org/10.1007/978-3-030-03044-5_10
8. 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>

9. Dauer, J.C., Finkbeiner, B., Schirmer, S.: Monitoring with verified guarantees. In: Feng, L., Fisman, D. (eds.) *Runtime Verification - 21st International Conference, RV 2021, Virtual Event, October 11-14, 2021, Proceedings. Lecture Notes in Computer Science*, vol. 12974, pp. 62–80. Springer (2021). https://doi.org/10.1007/978-3-030-88494-9_4
10. 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>
11. Falcone, Y., Krstic, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools Technol. Transf.* **23**(2), 255–284 (2021). <https://doi.org/10.1007/S10009-021-00609-Z>
12. Faymonville, P., Finkbeiner, B., Schirmer, S., Torfah, H.: A stream-based specification language for network monitoring. In: Falcone, Y., Sánchez, C. (eds.) *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings. Lecture Notes in Computer Science*, vol. 10012, pp. 152–168. Springer (2016). https://doi.org/10.1007/978-3-319-46982-9_10
13. 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
14. Finkbeiner, B., Keller, A., Schmidt, J., Schwenger, M.: Robust monitoring for medical cyber-physical systems. In: *Proceedings of the Workshop on Medical Cyber Physical Systems and Internet of Medical Things*. p. 17–22. MCPS '21, Association for Computing Machinery, New York, NY, USA (2021). <https://doi.org/10.1145/3446913.3460318>
15. Finkbeiner, B., Kohn, F., Scheerer, F., Schledjewski, M.: The RTLola Playground (2023), <https://rtlola.org/playground>
16. Finkbeiner, B., Kohn, F., Schledjewski, M.: Leveraging static analysis: An IDE for rtlola. In: André, É., Sun, J. (eds.) *Automated Technology for Verification and Analysis - 21st International Symposium, ATVA 2023, Singapore, October 24-27, 2023, Proceedings, Part II. Lecture Notes in Computer Science*, vol. 14216, pp. 251–262. Springer (2023). https://doi.org/10.1007/978-3-031-45332-8_13
17. 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. Lecture Notes in Computer Science*, vol. 12399, pp. 431–450. Springer (2020). https://doi.org/10.1007/978-3-030-60508-7_24
18. 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. Lecture Notes in Computer Science*, vol. 11237, pp. 282–298. Springer (2018). https://doi.org/10.1007/978-3-030-03769-7_16
19. 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>

20. Schwenger, M.: Monitoring cyber-physical systems: From design to integration. In: Deshmukh, J., Nickovic, D. (eds.) Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12399, pp. 87–106. Springer (2020). https://doi.org/10.1007/978-3-030-60508-7_5
21. Schwenger, M.: Statically-analyzed stream monitoring for cyber-physical systems (2022). <https://doi.org/10.22028/D291-37014>
22. Torfah, H.: Stream-based monitors for real-time properties. In: Finkbeiner, B., Mariani, L. (eds.) Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11757, pp. 91–110. Springer (2019). https://doi.org/10.1007/978-3-030-32079-9_6