Stream-based Monitoring under Measurement Noise*

Bernd Finkbeiner[1], Martin Fränzle[2], Florian Kohn(✉)[1], and Paul Kröger[2]

[1] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
{finkbeiner,florian.kohn}@cispa.de
[2] Carl von Ossietzky Universität, Oldenburg, Germany
{martin.fraenzle,paul.kroeger}@uol.de

Abstract. Stream-based monitoring is a runtime verification approach for cyber-physical systems that translates streams of input data, such as sensor readings, into streams of aggregate statistics and verdicts about the safety of the running system. It is usually assumed that the values on the input streams represent fully accurate measurements of the physical world. In reality, however, physical sensors are prone to measurement noise and errors. These errors are further amplified by the processing and aggregation steps within the monitor. This paper introduces RLola, a robust extension of the stream-based specification language Lola. RLola incorporates the concept of slack variables, which symbolically represent measurement noise while avoiding the aliasing problem of interval arithmetic. With RLola, standard sensor error models can be expressed directly in the specification. While the monitoring of RLola specifications may, in general, require an unbounded amount of memory, we identify a rich fragment of RLola that can automatically be translated into precise monitors with guaranteed constant-memory consumption.

Keywords: Slack Variables · Robust Monitoring · Measurement Noise

1 Introduction

Stream-based monitoring is a successful runtime verification approach for cyber-physical systems. Input streams containing sensor readings and other data are translated into output streams that process and aggregate this data. The resulting values on the output streams are then continuously evaluated against trigger conditions that characterize erroneous or dangerous situations. Tools for stream-based monitoring, like RTLola [4], TeSSLa [6] or Striver [14], are used in safety-critical applications, such as autonomous aircraft [3], where the precision of the monitoring result is vital. But how precise are stream-based monitors?

* This work was partially supported by the German Research Foundation (DFG) as part of PreCePT (FI 936/7-1; FR 2715/6-1) and of TRR 248 (No. 389792660), by the State of Lower Saxony within the Zukunftslabor Mobilität, and by the European Research Council (ERC) Grant HYPER (No. 101055412).

Most frameworks for stream-based monitoring operate under the assumption that the values on the input streams represent fully accurate measurements of the physical reality. As a result, the values in the output streams are also assumed to be precise, even if they are based on input data that was collected from physical sensors, which are prone to measurement noise and errors, and even if this data has been processed in a way that may have amplified these errors significantly.

A straightforward idea to keep track of the precision of the stream values is to lift the individual values from scalars to intervals, akin to interval-based robust monitoring of Signal Temporal Logic specifications [21]. Input streams then produce intervals centered around the measured value with a width defined by the precision of the sensor. The error in the output streams can be tracked via interval arithmetic: as more errors are combined into individual output values, the intervals of these outputs become larger, and we can determine whether a violation of the trigger conditions is possible, given the precision of the data.

Unfortunately, interval analysis usually leads to an overly pessimistic result. This phenomenon is known as the *aliasing* or *dependency problem*: in situations where errors cancel each other out, for example, because the same input is added and later subtracted from an aggregate value, interval arithmetic will still add, rather than subtract, the errors. In the trivial example, the term $x - x$ should evaluate to 0, independently of the value of x. However, if, because of noise, we assume x to be in the interval $[-10, 10]$, then interval arithmetic produces the even larger interval $[-10, 10] - [-10, 10] = [-20, 20]$.

In this paper, we present a new version of the Lola monitoring language [7], where we explicitly track the precision of the stream values. To avoid the aliasing problem, we introduce explicit *slack variables* that represent measurement noise by symbolically representing the interval $[-1, 1]$, which can be extended to arbitrary intervals through affine arithmetic [11]. Because each variable identifies a particular source of noise, they are not susceptible to the aliasing problem. In the example, the uncertain input is represented as $x + 10\epsilon$, where ϵ is a slack variable. Subtracting the input from itself would then result in $(x + 10\epsilon) - (x + 10\epsilon) = 0$.

The challenge in building a stream-based monitor with slack variables is the unbounded number of input values. If the noise on individual values of some input stream is at least partially independent, then we need a separate slack variable for each point in time. As slack variables are unlikely to resolve to scalar values, the number of slack variables in the equation store of the monitor may grow beyond any bound. In the paper, we demonstrate this phenomenon for the stream-based monitoring language Lola [7]. We define the syntax and semantics of RLola (*robust* Lola), the extension of Lola with slack variables.

It turns out, however, that for a large class of practically relevant RLola specifications, it is possible to combine multiple slack variables into a single variable without losing precision. For example, the term $x + 5\epsilon_1 + 5\epsilon_2$ is equivalent to $x + 10\epsilon'$ if ϵ_1 and ϵ_2 occurs only there. Suppose now that x is an output stream in which the term 5ϵ, with fresh slack variable ϵ, is added in every step. Then instead of keeping a growing term $5\epsilon_1 + 5\epsilon_2 + \ldots + 5\epsilon_n$ in memory, it suffices to count the number of steps n, and replace the n slack variables with a single slack variable with factor $5n$, resulting in the term $5n\epsilon'$. In fact, we can eliminate the slack variables and use an ordinary Lola specification to track the factor $5n$.

In the paper, we make use of this insight to identify a syntactic fragment of RLola for which we can automatically translate the given RLola specification into an equivalent Lola specification. The memory consumption of the resulting monitor is guaranteed to be constant.

1.1 Motivating Example

As a motivating example, consider an industrial warehouse robot that autonomously navigates by tracking its position through an unreliable indoor positioning system. To validate these potentially erratic position readings, a runtime monitor compares position measurements to positions computed from the traveled distance and a discrete direction. The movement directions of the robot are visualized in Figure 1. An RLola specification capturing this behavior is given in Example 1 below. Note that for simplicity, the computation of the y position is omitted as it follows analogously.

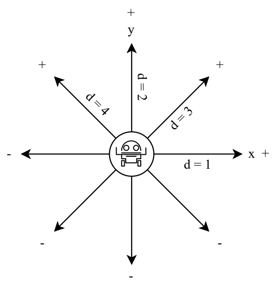


Fig. 1: Movement Directions in Example 1

Example 1 (Warehouse Robot).

```
input direction: Int
input raw_distance, position_x, position_y: Float
constant cos_45: Float := (1/2) * sqrt(2)

constant delta: Variable
output e: Variable
output distance := raw_distance + e + 5*delta

output computed_pos_x := if direction = 1 then
                      computed_pos_x.offset(by: -1, or: 0) + distance
                   else if direction = -1 then
                      computed_pos_x.offset(by: -1, or: 0) - distance
                   else if direction = 3 then
                      computed_pos_x.offset(by: -1, or: 0) + cos_45 * distance
                   else if direction = -3 then
                      computed_pos_x.offset(by: -1, or: 0) - cos_45 * distance
                   else if direction = 4 then
                      computed_pos_x.offset(by: -1, or: 0) - cos_45 * distance
                   else if direction = -4 then
                      computed_pos_x.offset(by: -1, or: 0) + cos_45 * distance
                   else
                      computed_pos_x.offset(by: -1, or: 0)

trigger position_x >_0.5 computed_pos_x || position_x <_0.5 computed_pos_x
```

Intuitively, slack variables extend the value domain of streams such that a stream of type `Variable` produces a fresh slack variable every time it is computed. As the precise value of the slack variables cannot be computed, such streams do not require a defining stream equation. Similarly, a constant of type `Variable` represents a single slack variable. In the above example, we define a stream of slack variables `e` and a single slack variable `delta` that are added to the `raw_distance` in the `distance` output stream to capture the measurement noise of the distance measurements. A more detailed description of this measurement error model is given in Section 2.2. Depending on the movement direction, this corrected distance is added proportionally to the computed position.

In the above example, all slack variables produced by the `e` stream will accumulate in the computed position streams, as the `offset(by: -1, or: 0)` operator refers to the last stream value (or 0 if that does not exist yet). As later shown in Section 5, multiple linear dependent slack variables can be combined without losing precision. Based on this, building a finite memory monitor for the above specification is possible. In fact, it is possible to translate the specification to one without slack variables:

```
input direction: Int
input raw_distance, position_x, position_y: Float
constant cos_45: Float := (1/2) * sqrt(2)

output d1 := if direction = 1
          then d1.offset(by: -1, or: 0.0) + 1.0
          else d1.offset(by: -1, or: 0.0)
output dn1 := if direction = -1
          then dn1.offset(by: -1, or: 0.0) + 1.0
          else dn1.offset(by: -1, or: 0.0)
output d3, dn3, d4, dn4 := ...

output delta_x := d1 - dn1 + d3*cos_45 - dn3*cos_45 - d4*cos_45 + dn4*cos_45
output epsilon_x := d1 + dn1 + d3 + dn3 + d4 + dn4
output raw_computed_pos_x := ...

output position_x_lower := raw_computed_pos_x - epsilon_x - delta_x
output position_x_upper := raw_computed_pos_x + epsilon_x + delta_x
trigger (position_x - position_x_lower) / (position_x_upper - position_x_lower) > 0.5
trigger (position_x_upper - position_x) / (position_x_upper - position_x_lower) > 0.5
```

Note that this specification is optimized by removing unreachable or equivalent cases. The `raw_computed_pos_x` is defined analogously to the `computed_pos_x` stream in Example 1 with the difference that it references the `raw_distance` input stream instead of the `distance` stream. Conceptually, this specification replaces the stream of slack variables `e` with eight individual slack variables, one for each direction. The streams `d1` to `d4` and their negative counterparts dynamically compute the coefficients of these slack variables, while the `delta_x` stream tracks the coefficient of the `delta` slack variable. The streams `position_x_lower` and `position_x_upper` reconstruct a precise lower and upper bound of the original `computed_pos_x` stream by applying interval arithmetic. In Section 5, we define a syntactical fragment of RLola for which such a transformation is always possible.

1.2 Related Work

In runtime verification, multiple logics have been studied to express valid system behavior, such as LTL [2], STL [19], or stream-based languages such as Lola [7], TeSSLa [6] or Striver [14]. In stream-based languages, the temporal history of measurements is represented as streams of values, and temporal properties are expressed through recursive stream equations involving time-offset accesses between these streams.

As established by Kauffman et al. [17], there are properties that cannot be monitored over unreliable channels that alter, delay, or lose data. While previous work was focused on missing or shifted events [18], this paper targets events that are present but mutated by measurement noise as specified by an error model.

While this could be encoded in first-order logic [8], the goal is to keep a strict memory bound on the resulting monitor. Kallwies et al. [16] handle missing events in the stream-based setting through symbolic input variables. The authors show that, in general, if streams are defined over real and boolean values, precise monitoring requires unbounded memory. Yet, this paper identifies a syntactic fragment in this domain that still allows for bounded memory monitors.

For temporal logics, there exist robust quantitative interpretations [10,9,13] that can handle inaccurate timestamps or measurement noise. Most such logics do not support measurement-related error models, leading to pessimistic verdicts. One exception is "truly robust" monitoring of Signal Temporal Logic [12], which also uses slack variables to express error models, but is significantly less expressive than stream-based specification languages.

2 Preliminaries

Runtime monitoring validates observed system behavior against a formal specification at runtime. The approach presented in this paper adapts the stream-based specification language Lola [7]. In the following, we provide an overview of its definition.

2.1 Lola

A Lola [7] specification consists of input streams, representing the observations made of the system, and output streams, which compute new values from input streams and other output streams. A Lola specification is a set of (recursive) equations over stream variables of the form:

$$o_1 = expr_1(i_1, ..., i_m, \quad o_1, ..., o_n) \quad \ldots \quad o_n = expr_n(i_1, ..., i_m, \quad o_1, ..., o_n)$$

where $o_1, ..., o_n$ are output stream variables and $i_1, ..., i_m$ are input stream variables and $expr_1, ..., expr_n$ are stream expressions.

Stream expressions determine how the next value of an output stream is computed. They are defined as arithmetic and logic expressions over stream variables. They include *if ... then ... else ...* clauses, stream variable references, and a stream offset operator: `.offset(by: i, or: l)` for $i < 0 \in \mathbb{Z}$ and some literal l. Further, we use `.last(or: l)` as syntactic sugar for `.offset(by: -1, or l)`.

The semantics of Lola is defined by an evaluation model that relates input stream values to output stream values. For its full definition, we refer to the original Lola paper [7]. Notice that a specification can have multiple valid evaluation models. For example, the specification: `output o = o` has infinitely many evaluation models for any given vector of input streams $\tau_1, \ldots, \tau_n$. Such specifications are called not well-defined due to their non-determinism. A specification is well-defined only if it assigns precisely one evaluation model to each vector of input streams $\tau_1, \ldots, \tau_n$. A syntactic criterion for well-definedness is given with the help of a dependency graph:

Definition 1 (Dependency Graph). *Let ϕ be a Lola specification. The dependency graph of ϕ is a directed weighted multi-graph $G = \langle V, E\rangle$ with $V = \{i_1, ..., i_m, o_1, ..., o_n\}$. An edge $e = \langle o_i, o_k, w\rangle$ is in E iff the expression of o_i contains o_k.offset(by: w, or: c) as a sub-expression (or $e = \langle o_i, i_k, w\rangle$ if i_k.offset(by: w, or: c) is a sub-expression). Analogously, edges with weight 0 are added for non-offset accesses.*

A specification is labeled *well-formed* iff its dependency graph does not contain any non-negative weight cycle, where the weight of a cycle is defined as the sum of all its edge weights.

2.2 Error Model

Following [12], we adapt the error model induced by the ISO norm 5725 [15] by decomposing the measurement error into a constant, but unknown per-sensor offset and a randomly varying per-measurement error. This decomposition directly correlates with the "trueness" and "precision" described in the ISO 5725 standard and is also reflected in Example 1 through the constant `delta` slack variable and the `e` stream of fresh slack variables. We adopt the definition of consistency from [12] and define when a series of sensor measurements is consistent with the unknown ground truth of a physical property.

Definition 2 (Consistency). *Let S be a sensor measuring a physical property at times $T \subseteq \mathbb{N}$ with a maximal sensor offset of $\delta \geq 0$ and a maximal random measurement error of $\epsilon \geq 0$. Let τ be the ground-truth time series. Then $m_S : T \to \mathbb{R}$ is a possible S* time series over *τ of sensor measurements iff*

$$\exists \varDelta \in [-\delta, \delta] : \forall t \in T : \exists \varepsilon \in [-\epsilon, \epsilon] : \tau(t) + \varepsilon + \varDelta = m_S(t).$$

We say the trajectory τ is consistent with *m_S and denote this fact by $m_S \models \tau$.*

Note that the consistency relation, as defined above, can be rewritten using affine arithmetic [11] as: $\tau(t) + \epsilon e_t + \delta d = m_S(t)$ if d is a slack constant and e_t is a per time-step fresh slack variable, ranging over the interval $[-1, 1]$.

3 Robust Lola

This section defines the syntax and semantics of RLola (*robust* Lola). RLola extends Lola with symbolic slack variables to represent error margins. To generate **slack variables** in RLola, we introduce the *Variable* value type.

An output stream of type *Variable* will produce a new slack variable in each time step or, in case of a constant stream, a single slack variable for the entire execution of the monitor.

```
input a_raw: Float

output e: Variable
constant d: Variable

output a := a_raw + 2 * e + 0.5 * d
```

Since slack variables are not explicitly bound to any values, streams of type *Variable* have no stream expressions. Instead, the variables symbolically represent a value in the range $[-1, 1]$. With streams of type *Variable* we can implement the measurement error model from Section 2.2 (like many other error models).

RLola Specifications. We define an RLOLA specification as a set of (recursive) equations over stream and slack variables as follows:

$$
\begin{aligned}
o_1 &:= \mathit{expr}_1(i_1, ..., i_I, \quad o_1, ..., o_O, \quad c_1, ..., c_C, \quad c_1^V, ..., c_{V_c}^V, \quad o_1^V, ..., o_{V_o}^V) \\
&\vdots \\
o_O &:= \mathit{expr}_O(i_1, ..., i_I, \quad o_1, ..., o_O, \quad c_1, ..., c_C, \quad c_1^V, ..., c_{V_c}^V, \quad o_1^V, ..., o_{V_o}^V)
\end{aligned}
$$

Where $i_1, ..., i_I$ are input stream variables, $o_1, ..., o_O$ are output stream variables, $c_1, ..., c_C$ are constants, $c_1^V, ..., c_{V_c}^V$ are constant slack variables, $o_1^V, ..., o_{V_o}^V$ are slack variable streams, $\mathit{expr}_1, ..., \mathit{expr}_O$ are stream expressions, and $c_1 := C_1, ..., c_C := C_C$ are constant streams with $C_1, ..., C_C \in \mathbb{R} \cup \mathbb{B}$.

Omitted from the definition above are *trigger streams*. They are defined as boolean output streams specifying assertions that are communicated to a system operator upon violation. *Stream expressions*, the *dependency graph* and *well-formed* specifications are defined as for Lola in Section 2.1.

RLola Semantics. As for Lola, the semantics for RLOLA is defined by an evaluation model connecting input stream values to output stream values. Let ϕ be a robust Lola specification with: input stream variables $i_1, ..., i_I$, output stream variables $o_1, ..., o_O$, constants $c_1, ..., c_C$, constant slack variables $c_1^V, ..., c_{V_c}^V$, and slack variable streams $o_1^V, ..., o_{V_o}^V$.

Let $\tau_1, ..., \tau_I$ be streams of length N of input values. Let $\sigma_1, ..., \sigma_O$ be streams of length N of output values. Let $\zeta_1, ..., \zeta_{V_c}$ be streams of length N of constant values. Let $\sigma_1^V, ..., \sigma_{V_o}^V$ be streams of length N of slack values. A single evaluation of ϕ is then defined as: $\psi := \{\tau_1, ..., \tau_I, \sigma_1, ..., \sigma_O, \sigma_1^V, ..., \sigma_{V_o}^V, \zeta_1, ..., \zeta_{V_c}\}$.

An evaluation model of ϕ is then the possibly infinite set of evaluations such that for all evaluations, the following holds:

$$
\begin{gathered}
\forall 1 \le t \le N, 1 \le i \le I, 1 \le o \le O, 1 \le o^V \le V_o, 1 \le c^V \le V_c : \\
\sigma_o(t) := \mathit{val}(\mathit{expr}_o)(t) \\
\sigma_{o^V}^V(t) \in [-1, 1] \\
\zeta_{c^V}(t) \in [-1, 1] \\
\zeta_{c^V}(t) = \zeta_{c^V}(t-1), \text{ if } t > 0
\end{gathered}
$$

Where $\mathit{val}(\mathit{expr}_o)(t)$ is defined for

$$o_o := \mathit{expr}_o(i_1, ..., i_I, \quad o_1, ..., o_O, \quad c_1, ..., c_C, \quad c^V_1, ..., c^V_{V_c}, \quad o^V_1, ..., o^V_{V_o})$$

with $c_j := C_j$ as follows:

$$\begin{aligned}
\mathit{val}(i_j)(t) &:= \tau_j(t)\\
\mathit{val}(o_j)(t) &:= \sigma_j(t)\\
\mathit{val}(c_j)(t) &:= C_j\\
\mathit{val}(c^V_j)(t) &:= \zeta_j(t)\\
\mathit{val}(o^V_j)(t) &:= \sigma^V_j(t)\\
\mathit{val}(f(\mathit{expr}_1, ..., \mathit{expr}_k))(t) &= f(\mathit{val}(\mathit{expr}_1)(t), ..., \mathit{val}(\mathit{expr}_k)(t))\\
\mathit{val}(\mathit{expr.offset}(\mathit{by{:}}\ i,\ \ \mathit{or{:}}\ l))(t) &= \begin{cases} \mathit{val}(\mathit{expr})(t+i) & \mathit{for}\ 1 \le t+i \le N \\ l & \text{otherwise} \end{cases}
\end{aligned}$$

An RLola monitor for a specification ϕ with the evaluation model φ and input streams $i_1, ..., i_I$ given the uncertain series of measurements $m_1, ..., m_I$ computes a (symbolic) representation of a set of evaluations

$$\varphi' := \{m_1, ..., m_I, \sigma_1, ..., \sigma_O, \sigma^V_1, ..., \sigma^V_{V_o}, \zeta_1, ..., \zeta_{V_c}\}$$

for any set of output streams $\sigma_1, ..., \sigma_O$, streams of slack values $\sigma^V_1, ..., \sigma^V_{V_o}$, and constant slack values $\zeta_1, ..., \zeta_{V_c}$ such that $\varphi' \subseteq \varphi$. Triggers are then evaluated existentially based on the set ψ'.

Boolean Conditions. With slack variables, stream equations may resolve to symbolically represented intervals instead of scalar values. We use the ternary predicates $>_p$ and $<_p$ to compare intervals to scalar thresholds in relation to an overlap percentage p. The overlap percentage sets a bound on the overlap of the interval with the threshold such that the predicate still evaluates to true:

The predicate $\mathit{expr}_i >_p c$ is satisfied for $p \in [0, 1]$ if the stream expression expr_i resolves to range $[l, u]$ and $(u - c)/(u - l) > p$ holds. The definition for $<_p$ is analogous.

The explicit definition of an overlap percentage concretizes the *inconclusive* verdict found in other logics with robust semantics (cf. [12,9]). There, comparing an interval of values to a scalar threshold produces an inconclusive verdict if the interval overlaps the threshold. The $>_p$ and $<_p$ predicates allow for a more precise assessment of the overlap.

4 Approximate Online Monitoring

We first present an online monitoring algorithm for RLola that over-approximates the semantics presented above. An evaluation algorithm for RLola has to manage two potentially unbounded quantities: the number of equations the monitor has to keep in memory and the length of these equations.

The number of equations in memory can be unbounded as, in general, the presented approach allows for refining or correcting earlier verdicts of the monitor at a later point in time, as slack variables can temporally relate measurement errors. Consider the scenario where the robot from Example 1 does not move, repeatedly measuring the same position. Such repeated measurements of the same physical value can increase the accuracy of the measured value, potentially leading to changes in past triggers. Yet, as shown in [12], to refine or correct previous monitor verdicts, the monitor would have to keep all previous stream equations in memory and solve a system of linear equations to evaluate trigger conditions across multiple time points. We argue that the monitor's constant memory footprint is more important than refining previous verdicts in an online monitoring setting, allowing the monitor to evict old stream equations.

The length of the equations can grow beyond any bound if more and more slack variables accumulate. Consider the following example. By definition, the equation for `sum` at time 3 includes the slack variables e_1, e_2, and e_3. As time progresses, the equation for `sum` grows, accumulating more and more slack variables. One approach to evaluate RLola specifications is to immediately interpret slack variables as the interval $[-1, 1]$.

```
input a_raw: Float
output e: Variable
constant d: Variable

output a := a_raw + 2 * e + 0.5 * d
output sum := sum.last(or: 0) + a
```

Interval Arithmetic. Interval arithmetic [20] lifts arithmetic operations such as addition and subtraction to intervals. An uncertain scalar value x can be represented as an interval $[a, b]$ of all possible values that x might have. An *interval* is defined as a set of real values $[a, b] = \{x \mid a \leq x \leq b\}$. Arithmetic operations and functions are then defined as follows: For two intervals $a = [a_l, a_u]$ and $b = [b_l, b_u]$ it holds that:

$$a + b = [a_l + b_l, a_u + b_u]$$
$$a - b = [a_l - b_u, a_u - b_l]$$
$$a * b = [\min(a_l b_l, a_l b_u, a_u b_l, a_u b_u), \max(a_l b_l, a_l b_u, a_u b_l, a_u b_u)]$$
$$a/b = a * \frac{1}{b} \text{ with } \frac{1}{b} = [{b_u}^{-1}, {b_l}^{-1}], \quad \text{if } 0 \notin b$$

In general, for any monotonic operation $\cdot$ it holds that:

$$a \cdot b = [\min(a_l \cdot b_l, a_l \cdot b_u, a_u \cdot b_l, a_u \cdot b_u), \max(a_l \cdot b_l, a_l \cdot b_u, a_u \cdot b_l, a_u \cdot b_u)]$$

Interval Approximation. In the following, we present a monitoring procedure that over-approximates the semantics of the RLola specification by translating it to a Lola specification defined over intervals using interval arithmetic. Note that Lola is generic regarding the value domains of streams and their supported operators, which enables this translation.

Interval Replacement. Let ϕ be an RLola specification with input stream variables $i_1, ..., i_I$, output stream variables $o_1, ..., o_O$, constants $c_1, ..., c_C$, constant slack variables $c_1^V, ..., c_{V_c}^V$, slack variable streams $o_1^V, ..., o_{V_o}^V$ and expressions $expr_1, ..., expr_O$. Let ϕ' be a Lola specification with expressions $expr'_1, ..., expr'_O$, where $expr'_i$ is equal to $expr_i$ with all references to $c_1^V, ..., c_{V_c}^V, o_1^V, ..., o_{V_o}^V$ replaced with the interval $[-1, 1]$. Boolean conditions, such as trigger conditions, are evaluated using the ternary operators defined in Section 3.

Proposition 1. *Let ϕ be an* RLola *specification and ϕ' be the Lola specification obtained from ϕ using interval replacement. Let ψ be the evaluation model of ϕ and ψ' be the evaluation model of ϕ'. If it holds for all sub-expressions of ϕ of the form*

$$\textit{if } p \textit{ then } expr_1 \textit{ else } expr_2$$

that p does not (transitively) reference any slack variable, then it holds for fixed streams of input data $\tau_1, ..., \tau_I$ that if $\{\tau_1, ..., \tau_I, \sigma_1, ..., \sigma_O, \sigma_1^V, ..., \sigma_{V_o}^V, \zeta_1, ..., \zeta_{V_c}\} \in \psi$ then $\{\tau_1, ..., \tau_I, \sigma_1, ..., \sigma_O\} \in \psi'$.

The proposition states that *interval replacement* indeed produces a Lola specification that over-approximates an RLola specification. Intuitively, intervals reintroduce the aliasing problem, which, as described in Section 1, results in over-approximating measurement noise. The additional syntactic requirement on *if* conditions stems from conditionals being non-monotonic functions.

5 Precise Constant Memory Online Monitoring

We now present a syntactic fragment of RLola for which constant-memory monitors exist. We develop this result in multiple steps. First, we define two requirements that all specifications in the fragment must fulfill and show how slack variables can be pruned from the monitor if their coefficients are linearly dependent. Then, we give examples of increasing complexity, highlighting how these requirements ensure the collinearity of subsets of the slack variables.

Requirement 1. First, we syntactically limit stream expressions. They are required to be in one of the following two forms:

$$expr_i := c_s * o_i[o, d] + c_i^T \begin{pmatrix} i_1 \\ \vdots \\ i_I \end{pmatrix} + c_\epsilon^T \begin{pmatrix} o_1^V \\ \vdots \\ o_{V_o}^V \end{pmatrix} + c_\delta^T \begin{pmatrix} c_1^V \\ \vdots \\ c_{V_c}^V \end{pmatrix} \quad (1)$$

$$expr_i := \text{if } p \text{ then } expr_i^c \text{ else } expr_i^a \quad (2)$$

Where $c_s \in \{0, 1\}$, p is a boolean stream expression and $c_i \in \mathbb{R}^I, c_\epsilon \in \mathbb{R}^{V_o}, c_\delta \in \mathbb{R}^{V_c}$. Intuitively, this requirement ensures that at every point in time, each equation entailed by a stream expression is an affine form [11].

Requirement 2. Second, we require that every output stream only occurs in dependency loops of equal weight. Concretely, let ϕ be a specification with output streams $o_1, ..., o_O$ and let $G = (V, E)$ be the dependency graph of ϕ, then:

$$\forall 1 \leq i \leq O. \exists c_o \in \mathbb{Z}. \forall \langle o_i \xrightarrow{o_1} ... \xrightarrow{o_n} o_i \rangle \in E. \sum_{1 \leq r \leq n} o_r = c_o$$

Intuitively, this requirement ensures that equations in the equation store are affine. For example, it prohibits calculating the Fibonacci sequence.

One exception from these requirements are trigger streams. As their value can, by definition, not be used by other streams and they are hence stateless, they can express arbitrary boolean assessments over output streams.

Consider Example 1; If the `distance` output stream is inlined, all stream expressions in the specification satisfy Requirement 1. Requirement 2 is also satisfied, as `computed_pos_x` and `computed_pos_y` are both only part of self-loops with weight -1.

We now develop the construction of constant-memory Lola monitors that monitor specifications of the fragment without loss of precision. We first define a method to reduce the number of slack variables in equations.

Definition 3 (Slack Variable Pruning). *Let $\vec{\epsilon} \in [-1, 1]^j$ be a vector of slack variables and let $C \in \mathbb{R}^{k \times j}$ be a matrix of coefficients, then*

$$y = C(\epsilon_1, ..., \epsilon_j)^T$$

defines a zonotope over slack variables $\epsilon_1, ..., \epsilon_j$. To prune slack variables, we reduce the dimension of $\vec{\epsilon}$ by finding collinear column vectors of the matrix C to obtain an equivalent (pruned) representation of the zonotope:

$$y = C'(\epsilon_1, ..., \epsilon_l)^T$$

with $C' \in \mathbb{R}^{k \times l}$ for $l \leq j$. If two or more column vectors of C are collinear, it holds that $l < j$.

We use Definition 3 to prune variables from the equations the monitor maintains at runtime. For that, we define the state of a monitor as follows:

Monitoring State. Let ϕ be an RLola specification with output streams:

$$o_1 := expr_1 \quad ... \quad o_O := expr_O$$

A monitor manages two sets of equations called equation stores: R for resolved equations of the form: $\sigma_i(t) = c' + c_1\epsilon_1 + \cdots + c_s\epsilon_s$ and U for unresolved equations of the form $\sigma_i(t) = expr_i$. At time t for measurements $m_1, ..., m_I$ the following equations are added to R: $\tau_1(t) = m_1, ..., \tau_I(t) = m_I$ and $\sigma_1(t) = expr_1, ..., \sigma_O(t) = expr_O$ to U. If, through simplifications, equations from U become resolved, they are moved to R.

At any time point t, the sets U and R are called the monitoring state. We call the equations in R monitoring equations.

```
input a_raw: Float

output e: Variable
constant d: Variable

output a := a_raw + e + d
output sum2 := sum2.last(or: 0) + 2a
output sum3 := sum3.last(or: 0) + 3a
```

(a) The RLola specification with slack variables.

```
input a_raw: Float
constant e: Variable
constant d: Variable

output e_coeff :=
  e_coeff.last(or: 0) + 1
output sum2_raw :=
  sum2_raw.last(or: 0) + 2a_raw + 2d
output sum3_raw :=
  sum3_raw.last(or: 0) + 3a_raw + 3d
output sum_2 := sum2_raw + 2 * e_coeff * e
output sum_3 := sum3_raw + 3 * e_coeff * e
```

(b) The translated RLola specification without slack variables.

Fig. 2: An RLola specification where a slack variable occurs with different coefficients.

Different Coefficients. As a first example, consider the specification in Figure 2a and its partial monitoring state depicted below:

	1	2	3	4
sum2	$2e_1 + 2d$	$2e_1 + 2e_2 + 4d$	$2e_1 + 2e_2 + 2e_3 + 6d$	$2e_1 + 2e_2 + 2e_3 + 2e_4 + 8d$
sum3	$3e_1 + 3d$	$3e_1 + 3e_2 + 6d$	$3e_1 + 3e_2 + 3e_3 + 9d$	$3e_1 + 3e_2 + 3e_3 + 3e_4 + 12d$

The table depicts partial monitoring equations truncated to their slack variable part at time points one to four. As discussed in Section 4, one can see that the slack variables produced by `e` accumulate in the monitoring equations of `sum2` and `sum3`. Yet, when rewriting the equations at time four as vectors, omitting the constant slack variable d:

$$\begin{pmatrix} 2,2,2,2 \\ 3,3,3,3 \end{pmatrix} (e_1, e_2, e_3, e_4)^T$$

it is easy to see that pruning, as defined in Definition 3, can reduce the number of slack variables to one, as all column vectors of the matrix are collinear. In fact, this holds for every time step due to Requirement 1 which ensures that slack variables only occur with a constant coefficient in stream equations. Based on this, Figure 2b depicts an equivalent RLola specification using only constant slack variables.

Different Offsets. Next, consider the example where slack variables are used in streams that reference themselves with different offsets in Figure 3a. Note that the constant slack variable d is omitted for simplicity. Again, consider the partial monitoring equations for this specification in the table below:

	1	2	3	4
sum	$2e_1$	$2e_1 + 2e_2$	$2e_1 + 2e_2 + 2e_3$	$2e_1 + 2e_2 + 2e_3 + 2e_4$
eo_sum	$3e_1$	$3e_2$	$3e_1 + 3e_3$	$3e_2 + 3e_4$

```
input a_raw: Float
output e: Variable

output a := a_raw + e
output sum := sum.offset(by: -1, or: 0) + 2a
output eo_sum := eo_summ.offset(by: -2, or: 0) + 3a
```

(a) The RLola specification with slack variables.

```
input a_raw: Float
constant e_even: Variable
constant e_odd: Variable

output step := step.last(or: 0) + 1
output e_even_coeff := if step % 2 = 0 ∧ step % 1 = 0
                    then e_even_coeff.last(or: 0) + 1 else e_even_coeff.last(or: 0)
output e_odd_coeff := if step % 2 = 1 ∧ step % 1 = 0
                   then e_odd_coeff.last(or: 0) + 1 else e_odd_coeff.last(or: 0)

output sum_raw := sum_raw.last(or: 0) + a_raw
output eo_sum_raw := eo_sum_raw.offset(by: -2, or: 0) + a_raw
output sum := sum_raw + 2 * e_even_coeff * e_even + 2 * e_odd_coeff * e_odd
output eo_sum := if step % 2 = 0 ∧ step % 1 = 0
               then eo_sum_raw + 3 * e_even_coeff * e_even
               else eo_sum_raw + 3 * e_odd_coeff * e_odd
```

(b) The translated RLola specification without slack variables.

Fig. 3: A specification where slack variables accumulate under different offsets.

Because of the offset of -2, a slack variable e_i is added at either an even position *or* an odd position of `eo_sum`, never at both. This stems from the stream-based semantics of RLola. Because of this, we analyze these two cases separately. Consider the equations at time three and four in their vector representation:

$$\begin{pmatrix}2,2,2\\3,0,3\end{pmatrix}(e_1,e_2,e_3)^T \qquad\qquad \begin{pmatrix}2,2,2,2\\0,3,0,3\end{pmatrix}(e_1,e_2,e_3,e_4)^T$$

Both matrices can be pruned to reduce the number of slack variables to two. With the same argument as for different coefficients, this holds for all even and odd positions, respectively. We give an equivalent RLola specification that only uses constant slack variables in Figure 3b. Note that the if conditions in the specification can be simplified. Yet, they are kept as is to demonstrate how the construction scales to arbitrary offsets in multiple streams as long as Requirement 2 is satisfied. In general, if a slack variable appears in multiple streams with different offsets $o_1, ..., o_k$, the above case distinction has to be extended to $s = o_1 * ... * o_k$ cases of offset combinations resulting in s constant slack variables.

Central Observation. When partitioning the monitoring equations of a specification in the fragment by case, the coefficients of the slack variables occurring in each case will always be equal. For example, in the above specification, the coefficients at even and at odd positions will always be equal. This is explained by the constant coefficients asserted by Requirement 1.

***If* Clauses.** Lastly, we extend this case distinction to *if* clauses. Consider the following example:

```
input a_raw: Float

output e: Variable
output a := a_raw + e
output sum := if a_raw > 10
            then sum.offset(by: -1, or: 0) + 2a
            else sum.offset(by: -1, or: 0) + 5a
output eo_sum := eo_sum.offset(by: -2, or: 0) + 3a
```

To handle *if* clauses, we distinguish one case per if condition. Let a slack variable output stream be referenced in n output streams that contain *if* conditions. Let there be a total of k if conditions in their stream expressions. Then, a monitor has to handle $s * (n + k)$ slack variables where s is the previous bound on the number of slack variables.

In the above example, there are two different offsets (-1 and -2), and one stream contains a total of one if condition. Hence, to precisely monitor the above specification, the monitor has to distinguish $2 * (1 + 1) = 4$ cases. In the following, we group the coefficients by their case:

	even	odd
a_raw ≤ 10	$\begin{pmatrix}5\\3\end{pmatrix} e_2$	$\begin{pmatrix}5\\3\end{pmatrix} e_1 + \begin{pmatrix}5\\3\end{pmatrix} e_5$
a_raw > 10	$\begin{pmatrix}2\\3\end{pmatrix} e_4$	$\begin{pmatrix}2\\3\end{pmatrix} e_3$

By Definition 3, the variables e_1 and e_5 of the above example can be pruned.

Following the central observation, it is easy to see that a monitor for the fragment only needs to keep a *single* constant slack variable for each case for each stream of slack variables. Figure 4 summarizes the construction.

The specifications generated by this construction can be evaluated, without any loss of precision, using the algorithm from Section 4. This is because all constant slack variables generated by the construction are only referenced once in streams that are *state-less*, meaning that they do not propagate through computations, preventing the aliasing problem.

Proposition 2. *The construction in Figure 4 is correct and only requires a bounded number of slack variables.*

As the specification is finite, there can only be a bounded number of cases; hence, the number of slack variables is also bounded. The correctness of the construction follows from Definition 3 and from the fact that, by the requirements of the fragment, a slack variable can only occur in one case per output stream at each time step.

Let $C_1, ..., C_k$ be the cases in which subsets of slack variables occur with equal coefficients in the monitoring equations.

1. For each case C_i introduce a constant slack variable d_i and construct a stream that counts how often this case occurs as follows:

```
constant d_i: Variable
output c_i := if C_i then c_i.offset(by:-1, or:0) + 1 else c_i.offset(by:-1, or:0)
```

2. For each output stream s and slack variable stream ϵ, where s references ϵ in cases $C_i, ..., C_j$ with coefficients $x_i, ..., x_j$ add a stream that reconstructs ϵ for s:

```
output s_ϵ := c_i * x_i * d_i + ... + c_j * x_j * d_j
```

3. For each output stream s and constant slack variable δ, where s references δ in cases $C_i, ..., C_j$ with coefficients $x_i, ..., x_j$ add a stream that reconstructs δ for s:

```
output s_δ:= (c_i * x_i + ... + c_j * x_j) * δ
```

4. For each output stream s that references slack variable streams $\epsilon_1, ..., \epsilon_n$ and constant slack variables $\delta_1, ..., \delta_m$ construct a stream s_raw that is equal to s, apart that all references to slack-variables are removed. Construct a stream s' that reconstructs s from its partial sums.

```
output s' := s_raw + s_ϵ1 + ... + s_ϵn + s_δ1 + ... + s_δm
```

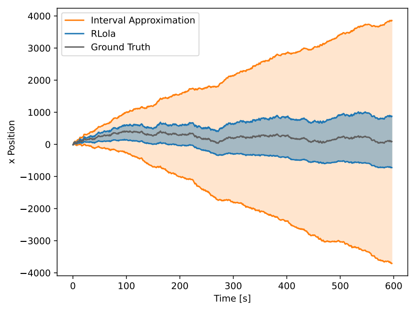
Fig. 4: Construction of constant-memory monitors.

6 Evaluation

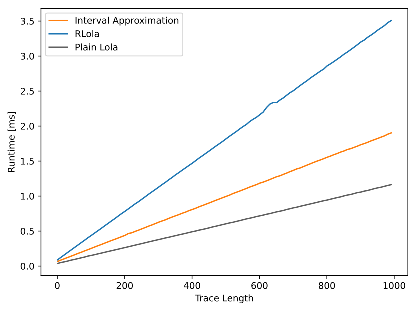
We evaluate our approach with respect to runtime and precision based on Example 1. We consider three variants of the specification: First, a Lola specification that does not take measurement errors into consideration. Second, a specification using the interval-based over-approximation presented in Section 4. Third, the RLola specification translated to regular Lola using the construction presented in Section 5. The experiments were conducted using the RTLola interpreter [5] on a MacBook Pro from 2020. A *ground truth* trace was randomly generated together with a mutated variant based on the error model presented in Section 2.2.

Figure 5a shows the error margins computed using interval arithmetic and slack variables around the ground truth x coordinates over time. While both error ranges grow over time, the graph visualizes the pessimistic over-approximation of interval arithmetic caused by the aliasing problem. It occurs due to the constant error `delta` in the example. In RLola this constant error is subtracted when the robot moves in opposite directions, while it is added in interval analysis.

Figure 5b compares the runtime of the three specifications in relation to the trace length. The running times were measured using the statistics-driven benchmarking library Criterion[1]. It shows that the plain Lola specification computes fastest, the interval-based specification is the second fastest, and the



(a) Precision of the computed x-position for the interval approximation vs. the precise evaluation of the slack variables.



(b) Monitor run-time of the plain Lola specification, the interval approximation, and the RLola specification.

Fig. 5: Precision and running time of RLola.

slack variable based specification has the highest computation time, taking 3.5 milliseconds for the whole trace of 1000 events. This is explained by the different number of auxiliary streams required to track the measurement errors in the different approaches. The performance could be improved by adding native support for the underlying computational domains to the RTLola interpreter.

7 Conclusion

We have presented RLola, a robust extension of the monitoring framework Lola. In RLola, the addition of slack variables allows us to track measurement noise induced by inaccurate sensors throughout computations; boolean verdicts explicitly account for the resulting inaccuracies. We demonstrated that RLola monitors require, in general, an unbounded amount of memory. We addressed this issue with a complete, but approximate, online monitoring algorithm based on interval arithmetic and a construction of fully precise constant-memory monitors for a rich fragment of RLola. Lastly, we demonstrated effectiveness by evaluating the above methods with respect to running time and precision.

This paper has focused on *online* monitoring, where data is processed in real time, and where trigger conditions are evaluated based on the information available during runtime. An interesting question for future work is how to adapt this approach to *offline* monitoring. In offline monitoring, it is possible to evaluate trigger conditions with the benefit of hindsight: measurements that were obtained only *after* a certain condition was evaluated may still allow for a more precise re-analysis of the trigger value. At the same time, resource constraints are less of a concern for offline monitoring. In online monitoring, the bounded-memory guarantee is crucially important so that the monitor can run indefinitely on constrained hardware. In offline monitoring, it is generally affordable to re-evaluate trigger conditions using SMT-solving. A similar approach was recently proposed for the offline monitoring of robust Signal Temporal Logic [12].

References

1. Aparicio, J., Heisler, B.: Criterion - statistics-driven micro-benchmarking library (2023), https://crates.io/crates/criterion
2. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14:1–14:64 (2011). https://doi.org/10.1145/2000799.2000800
3. Baumeister, J., Finkbeiner, B., Kohn, F., Löhr, F., Manfredi, G., Schirmer, S., Torens, C.: Monitoring unmanned aircraft: Specification, integration, and lessons-learned. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification. pp. 207–218. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-65630-9_10
4. Baumeister, J., Finkbeiner, B., Schirmer, S., Schwenger, M., Torens, C.: RT-Lola 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., Kohn, F., Oswald, S., Schledjewski, M., Schwenger, M., Scheerer, F., Stenger, M., Tentrup, L.: RTLola interpreter - rust crate on crates.io (2023), https://crates.io/crates/rtlola-interpreter
6. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: Tessla: Temporal stream-based specification language. In: Massoni, T., Mousavi, M.R. (eds.) Formal Methods: Foundations and Applications - 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26-30, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11254, pp. 144–162. Springer (2018). https://doi.org/10.1007/978-3-030-03044-5_10
7. 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
8. Decker, N., Leucker, M., Thoma, D.: Monitoring modulo theories. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8413, pp. 341–356. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_23
9. Donzé, A., Ferrère, T., Maler, O.: Efficient robust monitoring for STL. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 264–279. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_19
10. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: Chatterjee, K., Henzinger, T.A. (eds.) 8th International Conference on Formal Modeling and Analysis of Timed Systems, FORMATS 2010, Klosterneuburg, Austria, September 8-10, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6246, pp. 92–106. Springer (2010). https://doi.org/10.1007/978-3-642-15297-9_9

11. de Figueiredo, L.H., Stolfi, J.: Affine arithmetic: Concepts and applications. Numer. Algorithms **37**(1-4), 147–158 (2004). https://doi.org/10.1023/B:NUMA.0000049462.70970.B6
12. Finkbeiner, B., Fränzle, M., Kohn, F., Kröger, P.: A truly robust signal temporal logic: Monitoring safety properties of interacting cyber-physical systems under uncertain observation. Algorithms **15**(4), 126 (2022). https://doi.org/10.3390/A15040126
13. Fränzle, M., Hansen, M.R.: A robust interpretation of duration calculus. In: Hung, D.V., Wirsing, M. (eds.) Theoretical Aspects of Computing - ICTAC 2005, Second International Colloquium, Hanoi, Vietnam, October 17-21, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3722, pp. 257–271. Springer (2005). https://doi.org/10.1007/11560647_17
14. 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
15. ISO: ISO/IEC 5725:2023: Accuracy (trueness and precision) of measurement methods and results - Part 1: General principles and definitions. International Organization for Standardization, Geneva, Switzerland (July 2023)
16. Kallwies, H., Leucker, M., Sánchez, C.: Symbolic runtime verification for monitoring under uncertainties and assumptions. In: Bouajjani, A., Holík, L., Wu, Z. (eds.) Automated Technology for Verification and Analysis - 20th International Symposium, ATVA 2022, Virtual Event, October 25-28, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13505, pp. 117–134. Springer (2022). https://doi.org/10.1007/978-3-031-19992-9_8
17. Kauffman, S., Havelund, K., Fischmeister, S.: What can we monitor over unreliable channels? Int. J. Softw. Tools Technol. Transf. **23**(4), 579–600 (2021). https://doi.org/10.1007/S10009-021-00625-Z
18. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Thoma, D.: Runtime verification for timed event streams with partial information. 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. 273–291. Springer (2019). https://doi.org/10.1007/978-3-030-32079-9_16
19. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3253, pp. 152–166. Springer (2004). https://doi.org/10.1007/978-3-540-30206-3_12
20. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis. SIAM (2009). https://doi.org/10.1137/1.9780898717716
21. Visconti, E., Bartocci, E., Loreti, M., Nenzi, L.: Online monitoring of spatio-temporal properties for imprecise signals. In: Arun-Kumar, S., Méry, D., Saha, I., Zhang, L. (eds.) 19th ACM-IEEE International Conference on Formal Methods and Models for System Design, Virtual Event, China, November 20 - 22, 2021. p. 78–88. ACM (2021). https://doi.org/10.1145/3487212.3487344, https://doi.org/10.1145/3487212.3487344