From Active to Passive Monitoring: A Scheduling Approach with RTLola

Saarland University

Department of Computer Science

Master's Thesis

submitted by

Frederik Scheerer

Saarbrücken, January 2024

Supervisor:	Prof. Bernd Finkbeiner, Ph.D.
Advisor:	Jan Baumeister, M.Sc.
Reviewer:	Prof. Bernd Finkbeiner, Ph.D.
	Prof. Dr. Jan Reineke
Submission:	January 10, 2024

Abstract

Errors in cyber-physical systems often have fatal consequences. One effective approach to ensure that these systems behave correctly in all cases is runtime monitoring. With runtime monitoring, a monitor continuously checks a specification against the behavior of a system during operation.

However, a problem that most monitoring approaches overlook is the assumption that the bandwidth between the system and the monitor is infinite. In reality, the bandwidth is limited because the system might have various restrictions, for example in terms of energy consumption in an autonomous aircraft. Although bandwidth is limited, sensors of an autonomous aircraft will always produce new values in the same frequency, regardless of their immediate necessity. To make the best use of limited bandwidth, it would be beneficial to prioritize and query sensors more often which are more essential at any given time.

In this thesis, we investigate this idea and show our approach with RTLola. We propose a modification to the RTLola monitoring language so that inputs do not constantly receive new values from sensors. Instead, the monitor queries the sensors for new values when needed. To enable the monitor to request values, we allow the user to add timing information to the specification. We analyze the specification and calculate the appropriate query interval for each input at any given point in time. Finally, we evaluate this scheduled monitoring approach and compare it to the original version of RTLola.

Acknowledgements

I am very grateful to Prof. Bernd Finkbeiner for granting me the opportunity to pursue this master's thesis on such an interesting topic. I also wish to express my sincere thanks to Jan Baumeister for his invaluable guidance and constructive feedback as my advisor. Furthermore, I would like to thank Prof. Bernd Finkbeiner and Prof. Jan Reineke for taking the time to review this thesis.

I extend my appreciation to Jan Kautenburger for introducing me to the AirSim simulator and Florian Kohn for lending me his controller to navigate the simulation. Lastly, I would like to express gratitude to my family and friends for their ongoing support. Particularly, I want to thank Gideon, Jasmin and Simon for proofreading this thesis and giving valuable feedback. A special thank you to my partner Jasmin for her continuous support and encouragement.

Eidesstattliche Erklärung
Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath
I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung
Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent
I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 10 January, 2024

Erklärung
Ich erkläre hiermit, dass die vorliegende Arbeit mit der elektronischen Version übereinstimmt.

Statement
I hereby confirm the congruence of the contents of the printed data and the electronic version of the thesis.

Saarbrücken, 10 January, 2024

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cyber-physical systems are becoming more and more important and nowadays a lot of them behave completely autonomously. The failure of such cyber-physical systems often has fatal consequences. A human observer would be too slow to notice an error in the system and most probably could not intervene in time. In autonomous settings, there is not even a human present that could intervene. Therefore those systems must behave correctly in all cases. Only testing the system is not sufficient because it does not guarantee that all cases are covered by the tests. Static verification on the other hand proves that all executions of the system are correct. Given that most systems are too complex for static verification, runtime monitoring is used instead to monitor the behavior of a system during runtime. In this way, only the current execution trace is considered and not all possible execution traces as in static verification. For runtime monitoring, the system is equipped with a monitor that constantly receives information about the current state of the system. The monitor checks this information against a specification, to notice as early as possible if the system does not behave as planned.

In many applications, the monitor will receive data from a lot of different sources. In autonomous vehicles or drones, for instance, there are a number of different sensors that supply data to the monitor. In such systems, the bandwidth between the monitor and the sensors often plays a critical role. This bandwidth is constrained and can not be increased arbitrarily due to factors like energy consumption, physical size, or weight. In scenarios where the available bandwidth proves insufficient to transfer all possible incoming data, one has to prioritize the data that is most important.

As it is not possible to receive the readings of all sensors at the highest possible frequency, one has to evaluate which sensor readings are needed at which frequency to make sure no violation of the specification is missed by the monitor while still adhering to the bandwidth constraints. A lot of monitoring approaches do not make these assumptions and expect every change in the system to produce an update to the monitor. Some approaches support monitoring by querying the state of the system

periodically, but these often expect the period of each sensor to be the same during the whole runtime of the system. This is often not ideal, since the best frequency to query a sensor also depends on the current state of the system. With a drone, for example, the altitude sensor readings are irrelevant for the monitor while the drone is landed on the ground. The available bandwidth could be used more efficiently, by querying the altitude sensor less frequently and filling the bandwidth by querying data that is more important for a landed drone, for example from a camera monitoring the surrounding ground. The surroundings of the drone on the other hand do not change very fast if the drone is high up in the sky, allowing for less frequent queries of the camera in such instances.

Despite the effectiveness of many existing monitoring tools, they often lack the capability to perform the scheduling of the inputs to the monitor. Recognizing the existence of sufficient monitoring approaches, we do not want to invent another one but rather augment existing monitoring approaches with such a scheduling mechanism. In this thesis, we develop a scheduler designed to interface between an existing monitor and the sensors. The scheduler decides when to query the sensors for new inputs and passes these values to the underlying monitor.

To demonstrate our approach, we employ the stream-based monitoring language RTLola as the underlying monitor. An RTLola monitor receives the current values from the sensors as input streams. Then, the specification defines output streams that compute new values based on current and past values of input- and other output streams. The specification defines triggers which consist of a boolean-valued expression that indicates a violation of the specification if it becomes true. Because the RTLola language uses stream expressions, it is straightforward and intuitive to express powerful specifications.

For our monitoring approach, we modify the RTLola language to include annotations that allow the scheduler to decide on a good schedule. We want the specification for the monitor to contain as little scheduling information as possible while the schedule is derived automatically as much as possible. The annotated specification is then converted to a conventional RTLola specification that includes additional output streams that are used by the scheduler to decide when to schedule the inputs. Since we have converted the specification to conventional RTLola, we can use any RTLola backend as the underlying monitor.

This setup introduces challenges that we have successfully addressed: With this setup, the monitor does not automatically receive new data from the sensors anymore. Rather the scheduler has the task of manually querying the sensors for new values. Therefore, the scheduler must take into account that querying the sensors takes some time. Additionally, it is important to be prepared if the requested sensor readings arrive too late or not at all. In this case, the scheduler informs the monitor and lets the monitor handle that situation.

This thesis is structured in the following way: In Chapter 2, we give all the necessary background information on the RTLola monitoring language. This chapter also introduces the syntax and semantics of RTLola specifications. Chapter 3 deals with the scheduled monitoring approach. We define the syntax added to the RTLola language and explain the semantics of scheduling annotations and the semantics of a scheduled RTLola monitor. Following that, we present details about the implementation of the approach in Rust in Chapter 4. We then evaluate our scheduled monitoring approach in Chapter 5, by both running a scheduled monitor in an offline setting on a pre-recorded trace, as well as in an online setting with a simulated drone. In both cases, we compare the performance of the scheduled approach with that of regular RTLola.

Chapter 2

Preliminaries

This thesis employs the stream-based monitoring language RTLola. In the following section, we provide the necessary background information on this language.

2.1. RTLola

In this section, we introduce the stream-based specification language RTLola [1]. It evolved from the monitoring language Lola, which Ben d'Angelo et al. [2] introduced in 2005. RTLola is complementing Lola with real-time features such as sliding windows. Also, while Lola requires all inputs to be synchronized by a global clock, RTLola allows the inputs to receive values completely independent from each other. This is a property that is often necessary when monitoring real-time systems.

The introduction of the RTLola language is structured into different sections. First, we introduce the concept of stream-based monitoring and then introduce the RTLola language through small examples. In Sect. 2.1.3 we introduce the type system of RTLola, while Sect. 2.1.4 provides an overview of the syntax of the language. Finally, Sect. 2.1.5 addresses the semantics of RTLola monitors while Sect. 2.1.6 gives an overview of the analysis of RTLola specifications.

Remark 2.1.1. *In this section, we will use the definitions and stick to the notation presented by Schwenger in [3]. However, as we only need and explain a subset of RTLola, we have adapted the definitions accordingly. For the complete definitions, including spawn- and close clauses, we refer the interested reader to [3].*

2.1.1. Stream-Based Monitoring

Stream-based monitoring languages are given by defining stream-equations. A *stream* is a finite series of values of a specific type. We differentiate between input streams,

Stream

Input Stream
Output Stream
Trigger

output streams and triggers. An *input stream* represents data coming from the monitored system as an input to the runtime monitor. During the runtime of the monitored system, the system informs the monitor about its current state by inserting new values in the corresponding input streams. Stream-equations define *output streams*, calculating new values based on the current, but also previous values of the input streams or other output streams. A *trigger* is a boolean stream-expression defining a violation of the specification. Once the stream-expression of a trigger evaluates to true, a violation of the specification has occurred.

2.1.2. RTLola Language

Consider the following RTLola specification for monitoring an autonomous aircraft as an example:

Example 2.1.1. The specification compares consecutive values from the altitude sensor of an autonomous aircraft and issues a violation when the values differentiate too much:

```
input altitude : Float64
output alt_diff := abs(altitude - altitude.offset(by:-1).defaults(to:0))
trigger alt_diff > 10 "altitude changed too quickly"
```

The specification defines one input stream called `altitude`. This input represents the altitude sensor of the autonomous aircraft. Each time the altitude sensor produces a new altitude reading, the monitor adds that reading as a new value to the input stream. Based on this input stream, the specification defines the output stream `alt_diff`. A value of `alt_diff` describes the absolute difference of the current and previous altitude values. The trigger defines a violation of the specification when that difference is greater than 10. When a violation occurs, the monitor issues a warning with the message given next to the trigger expression. △

Synchronous Access

In Example 2.1.1, we can see that in RTLola, stream expressions can access other streams in different ways. The access of the current `altitude` value is called *synchronous*. Synchronous accesses use the current value of the accessed stream and require both accessing and accessed stream to receive a new value at the exact same moment. In Sect. 2.1.3 we will go into detail about how the specification defines which streams to update at which time.

Offset Access
Default Value

The example also contains an *offset access* to the altitude input. With an offset access, not the newest value of a stream is used, but an older value instead. Because these previous values could potentially not exist, an offset access always has to include a default value. The *default value* is used instead of the actual value if the requested value does not exist. In RTLola, every time accesses are not guaranteed to be successful, a default value has to be given.

Asynchronous Access

A third kind of access in RTLola is an *asynchronous access* with `hold`. A hold access accesses the newest value of a stream, regardless of when that value was produced. This

is the difference from synchronous accesses, which require the value to be produced at the exact same moment the access happens.

To demonstrate an asynchronous access with `hold`, consider the following example.

Example 2.1.2. In the example, the user can order the monitor to take a snapshot of the current altitude:

```
input take_snapshot : Bool
output snapshots := if take_snapshot then
    altitude.hold().defaults(to:0)
else
    snapshots.offset(by:-1).defaults(to:0)
```

By giving a value of `true` to the `take_snapshot` stream, the user indicates to the monitor to store the current altitude. Whenever such a value arrives in the input stream, the monitor computes the new value of `snapshots` by taking the newest value of the altitude stream. However, we do not require the altitude to receive a new value at the exact same time when the user requests a snapshot.

The example also shows another feature of RTLola: conditionals. Given whether the value of `take_snapshot` is true or false, the stream either uses the newest altitude or repeats the last stored snapshot. △

2.1.3. Type System

In RTLola, each stream has two types. The *value type* is the type of the values that are inside the stream. Possible value types are among others `Bool`, `Float64`, `Int64` or `UInt64`, where `UInt64` for example stands for a 64-bit unsigned integer. The `altitude` stream in our example has a value type of `Float64` since the altitude sensor of the drone produces 64-bit floating point numbers that are inserted in the input stream.

Value Type

Additionally, each stream has a *pacing type*. The pacing type defines when the monitor calculates new values for that stream. We differentiate streams with two kinds of pacing types: time-driven and event-driven streams.

Pacing Type

All the streams defined in the examples so far are event-driven. The time when the monitor evaluates an *event-driven* stream depends on a set of input streams. When all the input streams in the pacing of an event-driven output stream receive a new value, the monitor calculates a new value for the output stream as well. The pacing type of an event-driven output stream is often inferred automatically by the kind of accesses in the defining stream expression. Since synchronous accesses require the accessed stream to receive a new value at the same time, this requires the pacing type of the accessing stream to not evaluate that stream, if the accessed stream is not evaluated as well. Offset accesses influence the pacing type in the same way.

Event-Driven Stream

It is also possible to annotate the event-driven pacing type explicitly, by providing a positive boolean formula. This positive boolean formula consists of an expression containing conjunctions and disjunctions of input stream names. Annotated next to a

stream definition, the formula signals the monitor to only evaluate that stream if all the input streams of a conjunction or at least one input stream of a disjunction receive a new value at the same time.

Example 2.1.3. The following example counts the number of altitude readings the altitude sensor produced since the start of the monitoring:

```
output count_altitude @altitude := count_altitude.offset(by:-1).defaults(to:0) + 1
```

For this stream the pacing type is given explicitly by the positive boolean formula `@altitude`. This indicates, that every time the altitude input receives a new value, the monitor evaluates the `count_altitude` as well. By defining the stream in this way, the monitor counts the number of values the altitude sensor produced since the start of the monitoring, by always incrementing the `count_altitude` stream by one, whenever the altitude stream receives a new value. △

Time-Driven Stream

In contrast to event-driven streams, a *time-driven* stream is evaluated with a fixed frequency. The frequency is either explicitly annotated or also automatically inferred by synchronous accesses to other time-driven streams.

To demonstrate time-driven streams, consider the following example:

Example 2.1.4. In the example, every second, the stream adds the newest altitude at that time as a new value.

```
output altitude_steps @1Hz := altitude.hold().defaults(to:0)
```

In the example, the stream is annotated with a frequency of 1 Hz. Therefore, this stream is evaluated once per second with the newest altitude at that time. An synchronous access would not be allowed here, since the pacing types of `altitude` and `altitude_steps` are not compatible with each other. △

Sliding Window

In time-driven streams, RTLola allows to aggregate over a *sliding window*. A sliding window aggregation consists of a duration and an aggregation function. When a sliding window is evaluated, all the values of a stream that arrived inside the given duration in the past, are aggregated with the aggregation function. Possible aggregation functions are among others `sum`, `count`, `min`, `max`, `average` or `integral`.

The following example demonstrates the use of a sliding window:

Example 2.1.5. In the example, the specification defines a new output stream holding the average of all altitude readings that arrived in a specific window:

```
output average_altitude @1Hz :=
    altitude.aggregate(over: 5s, using: average).defaults(to: 0.0)
```

As this output stream is time-driven with a frequency of 1Hz, the monitor computes every second a new average altitude value. Always over all the values that, at that time, arrived in the last 5 seconds in the `altitude` input stream. If no altitude reading arrived in the last 5 seconds, the default value 0.0 is used instead, as the average is not defined for an empty set. △

The positive boolean formula is a static filter to restrict when the monitor generates new values of a stream. Additionally to the static filter, it is also possible to give a *semantic filter* condition. With a semantic filter, before evaluating the actual stream-expression that defines the new value of a stream, the monitor evaluates a stream-expression deciding if the monitor should produce a new value or not. The monitor only produces a new value, if the filter condition evaluates to true. Otherwise, no new stream value is calculated.

Semantic Filter

To demonstrate the use of a semantic filter, consider the following example:

Example 2.1.6. The specification counts the number of altitude readings that are greater than 100:

```
output count_high_altitude
    eval when altitude > 100
         with count_high_altitude.offset(by:-1).defaults(to:0) + 1
```

Because of the given filter condition, the stream is only incremented by one if the altitude is greater than 100.

Note that because of the synchronous access to `altitude` in the filter condition, this stream has an implicit pacing type of `@altitude`. The monitor always checks the filter condition when a new altitude arrives, and then only computes a new value, when the altitude is greater than 100. △

Remark 2.1.2. *To give a filter condition, one has to use an eval clause instead of the definition used in the examples before. The definition with* `:=` *is just an abbreviation for an eval clause with a filter condition that is always true.*

It is also possible to give multiple eval clauses. With multiple eval clauses, the monitor checks each filter condition from top to bottom and evaluates the stream with the stream-expression whose filter condition matches first. If no filter condition evaluates to true, the monitor does not calculate a new value for that stream.

The following example demonstrates the use of multiple eval clauses:

Example 2.1.7. The example defines a trigger that checks, whether the drone is flying too low. However, we want to allow a smaller altitude if the drone is flying slowly and are more strict with the allowed altitude if the drone is flying fast:

```
output altitude_trigger
    eval when velocity > 50 with altitude < 10
    eval when velocity ⩽ 50 with altitude < 5
trigger altitude_trigger.hold().defaults(to:false) "altitude too low"
```

The example defines an output stream `altitude_trigger`, that is used as the filter condition of a trigger. When the velocity is greater than 50, the trigger checks whether the altitude is smaller than 10. When the velocity is smaller or equal to 50, the trigger checks whether the altitude is smaller than 5. △

2.1.4. Syntax

The syntax of our reduced subset of the RTLola language is given by a grammar displayed in the form of a railroad diagram in Fig. 2.1. A specification consists of a sequence of input stream definitions (defined by the nonterminal `input`), simplified output stream definitions (defined by the nonterminal `simpleoutput`), regular output stream definitions (defined by the nonterminal `output`) and trigger definitions (defined by the nonterminal `trigger`). All of which follow a similar syntax. Each definition starts with a keyword (*input*, *output* or *trigger*) to indicate its kind. Input and output stream definitions include a name to identify and access the stream, while triggers do not require a name, as they can not be accessed. Input stream definitions specify the value type of that stream, while output streams can automatically infer the value type by the defining stream expression. Triggers always have a boolean value type and therefore also do not include an explicit value type. Optionally, output stream definitions may include an annotated pacing type, while it is automatically inferred when omitted. Input stream definitions and trigger definitions do not specify a pacing type as it is obvious for input streams and always inferred for triggers.

Output streams consist of multiple eval clauses (represented by the nonterminal `evalc`), each introduced by the keyword *eval*. Each clause can optionally include a filter condition (introduced by *when*) and always includes the stream expression (introduced by *with*) of that clause. For output streams with only a single eval clause without a filter condition, a more concise syntax is available as defined by the `simpleoutput` nonterminal. In this case, the stream expression follows directly after an :=, eliminating the need for additional keywords.

The stream expression of a trigger is given directly after the *trigger* keyword. After the stream expression, a trigger optionally includes a string in quotation marks. If given, this string provides additional information to the monitor when the trigger activates.

In the following subsections, we explain nonterminals from the RTLola grammar that are intentionally omitted as explicit grammar rules to simplify the syntax diagram.

Expressions

The stream expressions used in the filter condition and the eval definition of output streams (referred to as the nonterminal `expression`) are typical expressions. They consist of various constants, including integers, floating point numbers, and boolean values.

In addition to constants, stream expressions can include the stream accesses explained in Sect. 2.1.2. Furthermore, binary operators can be applied to two valid expressions in infix form, along with the use of unary operators and parenthesis for grouping within expressions. Special about the syntax of RTLola expressions compared to that of many programming languages is that for a lot of operators, both the conventional syntax known from most programming languages (like `(a <= 0)&&(b != 0)`), but also

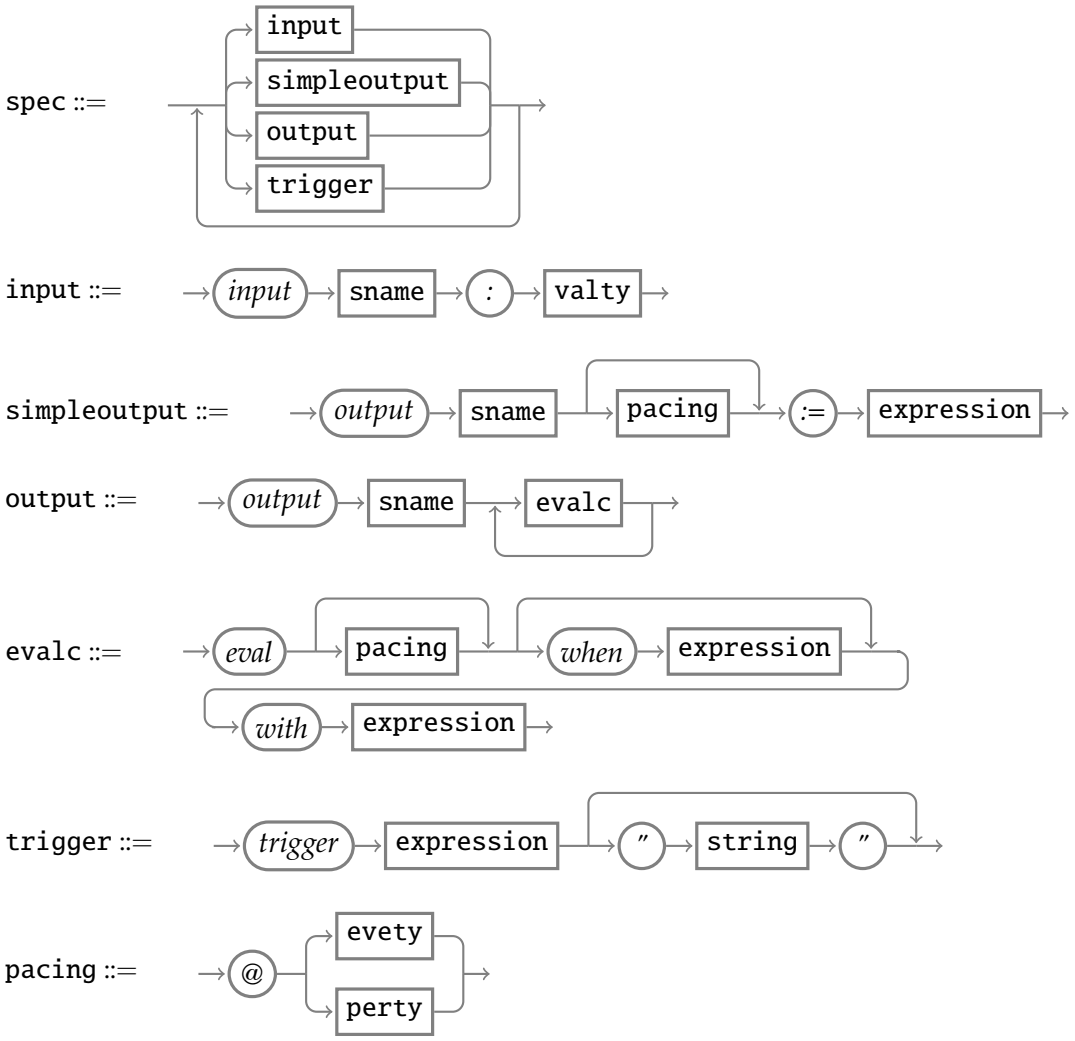


Figure 2.1.: The railroad diagram for the syntax of our subset of RTLola. A `nonterminal` node refers to the grammar rule called `nonterminal`, while a *terminal* node stands for the literal keyword *terminal*.

the mathematical operator as a Unicode symbol (like `(a ⩽ 0) ∧ (b ≠ 0)`) can be used. Additionally, built-in functions like `min`, `abs` or `sqrt` are supported as well.

Value Types

A value type (the nonterminal `valty`) represents a type of stream values. For signed integers, the value type is denoted as `Intx`, where `x` indicates the number of bits used to store that number. For instance, `Int64` describes a signed 64-bit integer. Unsigned integers with `x` bits follow a similar pattern and are expressed as `UIntx`. Likewise, floating point numbers exist as `Float32` and `Float64` depending on the required precision. Additionally, the type of boolean values is given by `Bool`.

Periodic Pacing Types

Periodic pacing types (the nonterminal `perty`) specify the period in which streams are evaluated. This information is specified by giving either a frequency or a period. Both kinds are given as a number with a corresponding unit. Frequencies are given in the unit Hertz (like `0.5Hz` or `2Hz`) and periods in time units (like `2s` or `500ms`). For example, a periodic stream that evaluates every two seconds could have a pacing type of either `@2s` or `@0.5Hz`.

Event-based Pacing Types

Event-based pacing types, denoted by the nonterminal `evety`, are constructed as a positive boolean formula where the literals correspond to the names of input streams. Essentially, event-based pacing types are the same as stream expressions but are exclusively composed of conjunction operators, disjunction operators, braces and input stream names.

2.1.5. Semantics

In this section, we will introduce the semantics of an RTLola monitor. An RTLola specification consists of a set of input streams denoted as Inputs and a set of output streams denoted as Outputs. For the semantic model, we depict triggers as additional boolean-valued output streams. Our specification consits of $n^{\downarrow} = |\text{Inputs}|$ input streams and $n^{\uparrow} = |\text{Outputs}|$ output streams. Additionally, the i'th input stream has the value type $\mathcal{T}_i^{\downarrow}$ and the j'th output stream has the value type $\mathcal{T}_j^{\uparrow}$.

The monitor semantics is defined by the asynchronous state-based real-time monitor function introduced by Schwenger [3]. In the following, we will introduce this function by providing a breakdown of each element in this lengthy term.

State-Based

The monitor function evaluates an RTLola monitor stepwise. Each time the monitor function is invoked, it computes one cycle of the monitor. In each cycle, each input stream can receive at most a single new value. After each cycle, the monitor function yields a verdict, which includes the current value of all output streams that received a new value during this cycle. Additionally, the monitor function updates an internal state. This internal state Σ^M is passed to each invocation of the monitor and contains past values of all streams. The past stream values are required for the evaluation of output streams that access streams with an offset. Furthermore, for sliding window aggregations, values from previous cycles are required as well.

The updated state that is returned by the monitor function for one cycle is then used with the next function invocation to compute the next cycle with new input values. As there is no previous state in the first cycle, an initial start state Σ_0^M is used. This state does not contain any previous values for each stream and also indicates that each sliding window has not started yet.

Asynchronous

In praxis, it is often not possible to guarantee that all input values arrive at the same time. Because of this, the monitor is asynchronous. This means, that not all inputs have to arrive at the same time and some inputs can arrive later or less often than others. For our semantics to model this behavior, the monitor function does not have to receive a value for every input. To model the possibility for an input stream to not receive a new value, we allow the monitor function to also receive the value $\bot$ as the new value of an input. A value of $\bot$ indicates, that no new value exists for that stream. The function therefore receives values from $\left\langle \mathcal{T}_i^{\downarrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\downarrow}}$, which is a sorted list of values of the corresponding value type for each input or $\bot$, if no such value exists.

Real-Time

An RTLola monitor offers support for real-time features, including periodic output streams and sliding windows. To enable these features, each value in each stream must have a timestamp attached. With this timestamp, for example, sliding windows can decide which values to include when aggregating over streams.

To facilitate this, the monitor function receives an additional input in the form of a positive real number, denoting the timestamp for the current input values. It is essential that these timestamps increase monotonically, meaning that for each cycle following the previous one, the timestamp must not decrease.

As a specification can contain periodic streams, it becomes necessary to evaluate them and calculate their new value when their defined periods have elapsed. To ensure that each periodic stream is evaluated at the correct time, the monitor function provides a

timestamp for the next periodic deadline. This timestamp signifies that the monitor must be evaluated at the latest once at that specified time to compute new values for periodic streams. If no input values arrive at that timestamp, the monitor calculates a cycle without any new inputs by giving $\langle \bot \rangle_{i \leqslant n^{\downarrow}}$ as an input and therefore only calculating new values for the periodic streams.

Definition

In the following, we will initially only provide the type signature of the monitor function. Subsequently, we give an example to help build an intuition about the function. Following this, we introduce auxiliary functions to be able to give the full definition of the monitor function in the end.

Definition 2.1 (Asynchronous State-Based Real-Time Monitor Function [3])

Def. Asynchronous State-Based Real-Time Monitor Function

Given some inputs $\left\langle \mathcal{T}_i^{\downarrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\downarrow}}$, a state Σ^M and a timestamp $\mathbb{R}^+$, the *asynchronous state-based real-time monitor function* computes one cycle of the monitor, returning a verdict $\left\langle \mathcal{T}_i^{\uparrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\uparrow}}$, an updated state Σ^M and the time of the next deadline $\mathbb{R}^+$:

$$M : \left\langle \mathcal{T}_i^{\downarrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\downarrow}} \times \Sigma^M \times \mathbb{R}^+ \to \left\langle \mathcal{T}_i^{\uparrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\uparrow}} \times \Sigma^M \times \mathbb{R}^+$$

Before defining the asynchronous state-based real-time monitor function M, we will introduce it with an example:

Example 2.1.8. For this example, consider the following specification:

```
input a : UInt64
input b : UInt64
output c @1Hz := a.hold().defaults(to:0)
output d := a + b
```

The specification consists of two inputs, `a` and `b`, and two outputs, `c` and `d`. The first output stream, `c`, is periodic and undergoes evaluation once every second. Assuming that the monitoring progress starts at time 1.0 s implies that the first evaluation of `c` takes place at 2.0 s, as that is the first time the period of `c` is passed since the start. This is followed by evaluations of `c` at 3.0 s, 4.0 s and so forth. The second output stream `d` on the other hand is event-based and has a pacing of `@(a&&b)`. Therefore, it computes new values, when both `a` and `b` receive new values at the same time.

Additionally, assume that the monitor receives the following input values:

t	a	b
1.0	2	4
1.7	6	$\bot$
3.0	1	3

In the table above, a value of $\bot$ indicates, that no new value exists for that input at that time.

The initial invocation of the monitor function may appear as follows:

$$M(\langle 2, 4\rangle, \Sigma_0^M, 1.0) = (\langle \bot, 6\rangle, \Sigma_1^M, 2.0)$$

The function call yields a verdict of $\langle \bot, 6\rangle$. This signifies that the first output stream, c, did not produce a new value, as it is periodic and first evaluated at time 2.0 s. In contrast, the output d computed a new value, the value 6. The function call also returns the updated state Σ_1^M, which incorporates the new values inside the state. Based on the output, it is also evident that the next call to correctly update the periodic output stream c must be executed at time 2.0 s. However, an input arrives at time 1.7 s, before the next periodic evaluation. Therefore, the subsequent call to the monitor function appears as follows:

$$M(\langle 6, \bot\rangle, \Sigma_1^M, 1.7) = (\langle \bot, \bot\rangle, \Sigma_2^M, 2.0)$$

This call does not produce any new output values, given that the first output is periodic and only due at time 2.0 s and the second output has a pacing of `@(a&&b)`, with b not receiving a new value in the current cycle, it is not evaluated as well. Nonetheless, the state is updated with the new values, accommodating potentially offset accesses, hold accesses or window aggregations.

Since no inputs arrive at time 2.0 s, it is necessary to issue an empty call to the monitor function to trigger the update of the periodic output stream c:

$$M(\langle \bot, \bot\rangle, \Sigma_2^M, 2.0) = (\langle 6, \bot\rangle, \Sigma_3^M, 3.0)$$

In this cycle, the monitor evaluates the output stream c and computes its new value, which is 6. As we have updated the state in the prior call, this information is included in the state Σ_2^M. The monitor does not evaluate the second output stream d since the pacing `@(a&&b)` is not satisfied with no new input values.

The next call to the monitor is at time 3.0 s because both the next input values and the next periodic deadline are set for time 3.0 s. The subsequent call, which marks the conclusion of this example, appears as follows:

$$M(\langle 1, 3\rangle, \Sigma_3^M, 3.0) = (\langle 1, 4\rangle, \Sigma_4^M, 4.0) \qquad \triangle$$

Before presenting the definition of the monitor function M, we introduce a set of auxiliary functions. Along with their usage in the definition of the monitor function M, these functions will reappear when we introduce the semantics of a scheduled RTLola monitor in Sect. 3.4. In this chapter, however, we do not provide the implementation details of these functions. We refer the interested reader to Schwenger [3] for the details.

It is crucial for the monitor to update periodic streams at their designated times, without missing any deadline. Therefore, the monitor has to know the precise moment at which the next periodic stream is due for evaluation. This knowledge allows the monitor to initiate an evaluation cycle at precisely the right time. The function nextdl accomplishes this task by computing the earliest time $\text{nextdl}(\sigma) \in \mathbb{R}^+$ at which the next periodic stream is due for evaluation, based on the current monitor state σ:

Definition 2.2 (Next Deadline [3])

Def. Next Deadline

Given a monitor state $\sigma \in \Sigma^M$, the *next deadline* $\text{nextdl}(\sigma)$ is the absolute point in time when the next periodic stream is due for evaluation:

$$\text{nextdl} : \Sigma^M \to \mathbb{R}^+$$

After each monitoring cycle, the monitor delivers a verdict: The new values of all output streams that received a new value during that cycle. This information is crucial for users to determine whether the monitor has identified any violations of the specification. Since the monitor state contains the history of all stream values, it naturally also contains the most current ones. To extract the verdict from the current monitor state, we employ the slice function:

Definition 2.3 (Slice [3])

Def. Slice

For a given monitor state $\sigma \in \Sigma^M$, the *slice of that state* $\text{slice}(\sigma)$ is a list of values for all output streams in the specification. If the stream produced a new value in the last cycle, the value for that stream corresponds to said new value. A value of $\perp$ indicates that no new value was calculated in the last cycle for that particular stream:

$$\text{slice} : \Sigma^M \to \left\langle \mathcal{T}_i^{\uparrow} \cup \{\perp\} \right\rangle_{i \leqslant n^{\uparrow}}$$

Stream expressions define output streams, which specify how new values are calculated for each stream. Moreover, streams only receive new values if their pacing type and any potential semantic filter conditions are met. The update function is responsible for managing all these operations: Given a set of input values, the update function calculates a new monitor state according to these input values.

Definition 2.4 (Update Function [3])

Given a monitor state $\sigma \in \Sigma^M$, a set of input values $ev \in \left\langle \mathcal{T}_i^{\downarrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\downarrow}}$ and a timestamp $t \in \mathbb{R}^+$, the *update function* calculates the new state $\sigma' \in \Sigma^M$. This new state reflects the monitor state after running one monitoring cycle with the new inputs *ev*:

Def. Update Function

$$\text{update} : \Sigma^M \times \left\langle \mathcal{T}_i^{\downarrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\downarrow}} \times \mathbb{R}^+ \to \Sigma^M$$

The update function accepts the current monitor state, the input values for the current cycle, and the current timestamp. Subsequently, it stores these new values in the monitor state and calculates and stores fresh values for all event-based output streams activated by the given inputs or periodic output streams that are due given the current time. Finally, the function returns the updated monitor state.

Leveraging these auxiliary functions, we can now define the monitor function M:

Definition 2.5 (Asynchronous State-Based Real-Time Monitor Function [3])

Given a set of input values $ev \in \left\langle \mathcal{T}_i^{\downarrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\downarrow}}$, a monitor state $\sigma \in \Sigma^M$, a time $t \in \mathbb{R}^+$ and $\sigma' = \text{update}(\sigma, ev, t)$, the *asynchronous state-based real-time monitor function* runs one cycle of the monitor with the given input values *ev*:

$$M : \left\langle \mathcal{T}_i^{\downarrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\downarrow}} \times \Sigma^M \times \mathbb{R}^+ \to \left\langle \mathcal{T}_i^{\uparrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\uparrow}} \times \Sigma^M \times \mathbb{R}^+$$

$$M(ev, \sigma, t) = (\text{slice}(\sigma'), \sigma', \text{nextdl}(\sigma'))$$

Initially, the update function is invoked to update the monitor state by executing one cycle with the provided inputs. The resulting updated monitor state σ' is subsequently utilized to extract the verdict $\text{slice}(\sigma')$ and compute the deadline for the next periodic output stream $\text{nextdl}(\sigma')$.

2.1.6. Analysis

Stream-based monitoring languages offer the advantage of being easily statically analyzed to retrieve important information such as the evaluation order, memory bounds of streams and automatic pacing type inference. The dependency graph serves as a foundation for extracting all of that information.

The dependency graph can be statically generated from an RTLola specification, describing the dependencies between streams in the specification. This directed graph establishes edges between streams, indicating stream accesses from one stream to another:

Definition 2.6 (Dependency Graph [3])

Def. Dependency Graph

The *dependency graph* $DG = (V, E)$ of an RTLola specification is a labeled, directed multigraph. Each stream in the specification corresponds to a vertex in the graph and each edge represents a dependency between two streams with $E \subseteq V \times L_D \times V$, where L_D describes the kind of dependency:

$$L_D = \underbrace{\{\mathit{Filter}, \mathit{Eval}\}}_{\text{Source}} \times (\{\mathit{Sync}\} \cup \underbrace{\{\mathit{Async}\}}_{\text{Hold}} \cup (\mathit{Offset} \times \mathbb{N}) \cup \underbrace{(\{\mathit{Async}\} \times \mathcal{F}_H \times \mathbb{R})}_{\text{Sliding Windows}})$$

where $\mathcal{F}_H$ describes the window aggregation function.
The edges in the graph are created according to the following rules:

- An input stream does not have outgoing edges.
- For an output stream, determine the edges and access kinds of the filter and eval expression. Label them with Filter or Eval respectively.
- For expressions occuring in an output stream or trigger σ, each synchronous access, hold access or offset access to σ' translates to an edge $(\sigma, \_, \sigma') \in E$, with access kind Sync, Async or (Offset, n) respectively.
- Each aggregating access to σ' with aggregation function $\gamma \in \mathcal{F}_H$ and duration $d \in \mathbb{R}$ translates to $(\gamma, \_, \gamma')$ with access kind $(\mathit{Async}, \gamma, d)$.

The set of resulting edges is E.

We will demonstrate the creation of a dependency graph for an RTLola specification with the following example:

Example 2.1.9. Consider the following RTLola specification:

```
input a : Int64
output b := a + c.offset(by:-1).defaults(to:0)
output c := b + a
trigger c > 0
output d eval when a > 0 with b
```

Fig. 2.2 depicts the dependency graph of this specification.
The input stream a does not have any outgoing edges. Each edge in the graph is labeled with Eval because each stream access in the specification happens in the eval expression. The only exception is the access from d to a: Because the access happens in the filter condition, the edge is labeled with Filter. △

Memory Bound

Leveraging the dependency graph, we can determine the *memory bound* of a stream. The memory bound describes the number of values the monitor must retain for a given

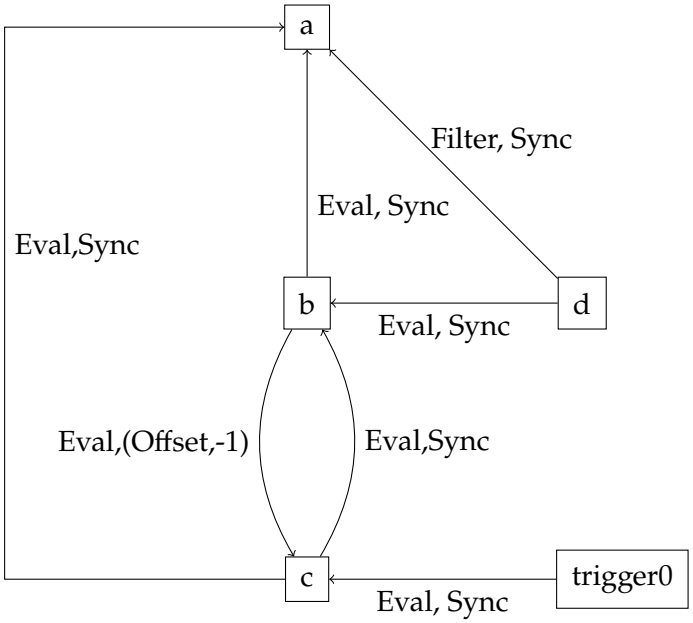


Figure 2.2.: The dependency graph for the specification in Example 2.1.9

.

stream to ensure the successful execution of the monitoring. In the context of the entire specification, this value is directly related to the amount of memory required for the monitor to execute. In our specific subset of the RTLola language, this memory bound remains static and can be computed from the dependency graph based on the largest offset access to a stream.

Furthermore, the dependency graph is utilized in automatically deriving pacing types for streams by considering all synchronous access to other streams within the specification.

Example 2.1.10. In the specification given above in Example 2.1.9, the memory bound of the output stream `b` is 1, as only the current value of the stream has to be stored. For the output stream `c`, the last two values are needed to accommodate for the offset access from `b`. Therefore, the memory bound of `c` is 2.

Based on the synchronous accesses to input `a`, the analysis infers a pacing type of `@a` for all output streams. △

Chapter 3

Scheduled RTLola

This chapter introduces scheduled RTLola. Scheduled RTLola complements the regular RTLola language with a mechanism to attach scheduling annotations in the form of constraints to individual streams and triggers. Based on these constraints, a scheduler decides which inputs to query at which moment in time. To make the scheduling approach more flexible, we allow the annotations to reason about different kinds of values, which differently determine which inputs to query at which time. The values that are used inside scheduling annotations are called constraint values and are introduced in Sect. 3.1. Following that, we introduce two different kinds of constraint values used in this work: durations and priorities. Subsequently, Sect. 3.2 introduces the syntax of annotated RTLola specifications.

We break down the semantics into two components to comprehend the workings of scheduled RTLola: the specification semantics and the monitor semantics. The specification semantics is dedicated to examining an annotated RTLola specification. It describes the steps taken to dissect the annotated specification, analyze the constraints and construct a schedule that adheres to all constraints in the specification. This topic is dealt with in Sect. 3.3.

The monitor semantics (Sect. 3.4) on the other hand takes a closer look at the semantics of the scheduled monitoring component. It describes the scheduled counterpart of the monitor function M as introduced in Def. 2.1. The scheduled version of the monitor function M operates on a state $\Sigma^{M,S} = (\Sigma^M, \Sigma^S)$, where Σ^M represents the monitoring state of the regular RTLola semantics containing past stream values and timing information. Additionally, the state contains the schedule state Σ^S, which carries information concerning the streams scheduled by the scheduler:

→ Def. 2.1, p. 14

Definition 3.1 (Schedule state)

The *schedule state* Σ^S stores information about the current scheduling of input streams:

Def. Schedule State

$$\Sigma^S = \mathcal{P}\left(\mathcal{P}\left(\text{Inputs}\right) \times \mathbb{R}^+ \times \text{Value}\right)$$

The schedule state maps sets of input streams $\mathcal{P}(\text{Inputs})$ to a timestamp $\mathbb{R}^+$ of their last update and the currently assigned constraint value.

3.1. Constraint Value

We support various kinds of schedulers that all follow a general approach. For all kinds of schedulers, the translation process remains the same, while only the specific values scheduled for each input are different. These specific values are called constraint values:

Def. Constraint Value

Definition 3.2 (Constraint Value)

Constraint values represent a set of values, denoted as Value, with the following properties:

1. Each set of constraint values is associated with a total order $\preceq \subseteq (\text{Value} \times \text{Value})$. Given two constraint values $a, b \in \text{Value}$, the ordering $a \preceq b$ implies that the scheduler, when faced with the decision of choosing between a and b as the scheduled value for an input stream, prioritizes the value b compared to a for that input.

2. Each set of constraint values provides a function

   $$\text{nextdl}^S : \Sigma^S \times \mathbb{R}^+ \to \mathbb{R}^+$$

   which given the current scheduling state $\Sigma^S = \mathcal{P}(\mathcal{P}(\text{Inputs}) \times \mathbb{R}^+ \times \text{Value})$ of the scheduled monitor function M and the current time $\mathbb{R}^+$, returns the absolute time of the *next scheduled input*.

3. Each set of constraint values provides a function

   $$\text{pop} : \Sigma^S \times \mathbb{R}^+ \to \Sigma^S \times \mathcal{P}(\text{Inputs})$$

   which given the current time $\mathbb{R}^+$ updates the current scheduling state Σ^S of the monitor function M and returns a *set of input streams that are due at the next scheduled deadline*.

These three requirements are sufficient to schedule with a given set of constraint values. In this work, we focus on two specific types of constraint values: duration and priorities. In the following sections, we will explain what these two kinds of constraint values describe.

3.1.1. Durations

The first kind of constraint values are durations. As time is represented as positive real numbers in the semantics of RTLola, the set of constraint values is

$$\text{Value} = \mathbb{R}^+.$$

The scheduler assigning a duration of $x \in \mathbb{R}^+$ to an input stream indicates that the input stream should receive a new value after a time interval of x time units. As a result, the annotated specification contains constraints about the minimum and maximum time that should lie between two consecutive updates of streams. The scheduler has the task of identifying suitable durations for individual inputs or combinations of them.

As duration constraint values are represented as positive real numbers, the scheduling state Σ^S is

$$\Sigma^S = \mathcal{P}\left(\mathcal{P}\left(\text{Inputs}\right) \times \mathbb{R}^+ \times \mathbb{R}^+\right).$$

The requirements for duration constraint values given in Def. 3.2 are defined as follows:

1. We define the total order $\preceq \subseteq \mathbb{R}^+ \times \mathbb{R}^+$ as

   $$a \preceq b \iff b \leqslant a \quad \forall a, b \in \mathbb{R}^+.$$

   If b represents a shorter duration than a, i.e. $b \leqslant a$, the scheduler should prioritize the duration b against the duration a. This is the case because when the scheduler has to decide between a and b as both suitable durations for an input, it should choose the smaller one.

2. The duration scheduled for an input stream specifies the relative duration after which the stream should receive a new value. The overall next deadline is then defined as the minimum over all scheduled inputs:

   $$\text{nextdl}^S : \Sigma^S \times \mathbb{R}^+ \to \mathbb{R}^+$$

   $$\text{nextdl}^S(\sigma^S, \_) = \min\left\{t + d \mid (\_, t, d) \in \sigma^S\right\}$$

   The schedule state consists of triple $(I, t, d) \in \mathcal{P}\left(\text{Inputs}\right) \times \mathbb{R}^+ \times \mathbb{R}^+$. For each set of inputs I, the duration d was scheduled at time t. The absolute deadline for an individual set of inputs is therefore the duration d added to the start time t. The next absolute deadline for all scheduled inputs is the minimum for each individual one. As the current time is irrelevant for this, the second argument is simply ignored for the definition of nextdl^S over duration constraint values.

3. The pop function removes all the inputs scheduled at the next scheduled deadline from the schedule and returns a set of all removed inputs. This list then contains all the inputs that are due at that deadline. For duration constraint values, we define the pop function as follows:

$$\text{pop} : \Sigma^S \times \mathbb{R}^+ \to \Sigma^S \times \mathcal{P}(\text{Inputs})$$
$$\text{pop}(\sigma^S, \_) = (\sigma^{S\prime}, S)$$

with

$$\sigma^{S\prime} = \left\{ (I, t, d) \in \sigma^S \mid t + d > \text{nextdl}^S(\sigma^S) \right\}$$
$$S = \left\{ i \in I \mid (I, t, d) \in \sigma^S / \sigma^{S\prime} \right\}.$$

The pop function modifies the state σ^S by removing all the inputs from the schedule, that are scheduled at the next scheduled deadline. Additionally, the function collects all those removed inputs and returns them as a set.

To demonstrate duration constraint values and the definition of the requirements, consider the following example:

Example 3.1.1. Consider the following schedule state $\sigma^S \in \Sigma^S$ for a specification with input streams a and b when scheduling duration constraint values:

$$\sigma^S = \{(\{a\}, 1, 3), (\{a, b\}, 2, 1), (\{b\}, 3, 2)\}$$

The state represents all currently scheduled input streams:

1. Input stream a is scheduled to receive a new value three time units after it was scheduled. This duration was scheduled at the absolute time 1.

2. Input streams a and b are scheduled to both receive a new value one time unit after they were scheduled at absolute time 2.

3. Input stream b is scheduled to receive a new value three time units after it was scheduled at time 2.

Therefore, independent of the current time $t \in \mathbb{R}^+$, the next deadline of that particular schedule state σ^S is

$$\text{nextdl}^S(\sigma^S, t) = 3.$$

This is the case because $\{a\}$ is scheduled at time $1 + 3 = 4$, $\{a, b\}$ are scheduled at time $2 + 1 = 3$ and $\{b\}$ is scheduled at time $3 + 2 = 5$ from which 3 is the minimum.

Popping all streams that are scheduled at the next absolute deadline at an arbitrary time $t \in \mathbb{R}^+$ returns the following scheduled input streams:

$$\text{pop}(\sigma^S, t) = (\sigma^{S\prime}, \{a, b\})$$

with a new scheduling state

$$\sigma^{S\prime} = \{(\{a\}, 1, 3), (\{b\}, 3, 2)\}.$$

In the returned scheduling state $\sigma^{S\prime}$, the triple $(\{a, b\}, 2, 1)$ was removed and the pop function returned the inputs a and b of that particular triple.

For the scheduler, this information indicates that at the next absolute deadline 3, the monitor is expecting to run a new monitoring cycle with new values for both inputs a and b. △

Scheduling based on durations offers the advantage of providing clear predictability regarding the monitoring progress. A schedule with durations is easy to comprehend and verify, which is crucial in the context of runtime monitoring. On the other hand, it can be challenging to specify constraints for a specification in terms of durations for individual streams. For the user designing the specification, it might be more straightforward to determine the importance of individual streams based on the current state of the monitor. This is possible with scheduling over priorities of streams instead of durations and is described in the following section.

3.1.2. Priorities

This approach describes a schedule based on priorities assigned to streams in the specification. With this method, a scheduler takes charge of determining which streams to update next instead of the user having to assign individual durations. The user's input is limited to specifying the maximum available bandwidth. With this information, the scheduler's role is to use the available bandwidth to ensures streams with a higher priority are processed first, while also guaranteeing that all inputs receive new values periodically.

Remark 3.1.1. *In this work, we consider the bandwidth as the number of input values processed within a specified time period. However, it would be possible to improve the effectiveness of this approach by considering that values with different value types also have varying sizes. In future work, it might be beneficial to expand the priority constraint values to allow expressing the bandwidth in terms of bits per time period, rather than just the number of values processed.*

In this approach, a priority is represented as a positive integer

$$\text{Value} = \mathbb{N}$$

where higher numerical values correspond to a higher priority. We would therefore describe the schedule state as:

$$\Sigma^S = \mathcal{P}\left(\mathcal{P}\left(\text{Inputs}\right) \times \mathbb{R}^+ \times \mathbb{N}\right).$$

The requirements for priority constraint values given in Def. 3.2 are defined as follows:

1. Priorities are represented as positive integers, while a low number represents a low priority and a high number represents a higher priority. As a result, if two inputs are scheduled and one has a higher priority than the other, the one with the higher priority is prioritized against the other. The total order follows from this property:

   $$a \preceq b \iff a \leqslant b \quad \forall a, b \in \mathbb{R}^+.$$

   If b has a higher priority than a, i.e. $a \leqslant b$, b should be prioritized over a.

2. This kind of priority constraint value is designed to utilize all the available bandwidth and fill it with the most valuable data depending on the current monitor state. When working with a bandwidth of $x \in \mathbb{N}$ values every $t \in \mathbb{R}^+$ time units, the monitor should be updated in intervals of t time units with x new values every cycle. Consequently, the next deadline occurs t time units after the current time t' independent from the current schedule state:

   $$\text{nextdl}^S : \Sigma^S \times \mathbb{R}^+ \to \mathbb{R}^+$$

   $$\text{nextdl}^S(\_, t') = t' + t$$

3. The primary objective is to ensure that the available bandwidth is allocated to inputs with the highest assigned priorities. However, it is essential to prevent any inputs from starving and never receiving any new value due to other streams constantly having higher priorities. To address this concern, we define a time duration $d \in \mathbb{R}^+$ during which each input should at least receive a single new value. Input streams that exceed this time duration without receiving a new value are considered "overdue streams". Given the current time $t \in \mathbb{R}^+$ and the time $s \in \mathbb{R}^+$ of the last update of a stream, we define it as overdue if it did not receive a new value in more than d time units:

   $$\text{overdue} : \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{B}$$

   $$\text{overdue}(s, t) = t - s \geqslant d.$$

   When selecting inputs to fill the available bandwidth, we aim to take both overdue streams and the assigned priorities into account. To achieve this, we introduce a

total order $\succeq'_t$ on the elements of the schedule state σ^S dependent on the current time $t \in \mathbb{R}^+$:

$$\succeq'_t \subseteq (\mathcal{P}(\text{Inputs}) \times \mathbb{R}^+ \times \mathbb{N}) \times (\mathcal{P}(\text{Inputs}) \times \mathbb{R}^+ \times \mathbb{N}).$$

Note that $\succeq'_t$ is distinct from the previously defined order $\preceq$, which is not an order on the elements of σ^S but rather on individual priority values.

The total order $\succeq'_t$ defines which scheduled inputs to prioritize against others. It is established using a lexicographical order with the following prioritization rules:

a) Overdue streams take precedence over streams that are not overdue.

b) Streams with a higher assigned priority take precedence over streams with lower priorities.

c) Streams that have not been updated for a longer duration are favored over streams that were updated more recently.

Formally, we define $\succeq'_t$ as follows:

$$\begin{aligned}(I, s, p) \succeq'_t (I', s', p') \iff & \text{ overdue}(s, t) \wedge \neg\text{overdue}(s', t) \\ & \vee \text{overdue}(s, t) = \text{overdue}(s', t) \wedge p' \preceq p \\ & \vee \text{overdue}(s, t) = \text{overdue}(s', t) \wedge p = p' \wedge s < s'.\end{aligned}$$

The pop function is responsible for selecting which inputs to allocate the bandwidth with. The function is designed to remove and allocate $x \in \mathbb{N}$ values from the schedule, with x being the number of values the bandwidth can accommodate. These x values are the first x inputs in the schedule state when ordering the scheduled inputs with the order $\succeq'_t$:

$$\text{pop} : \Sigma^S \times \mathbb{R}^+ \to \Sigma^S \times \mathcal{P}(\text{Inputs})$$

$$\text{pop}(\sigma^S, t) = (\sigma^{S\prime}, D)$$

with D corresponding to all inputs contained in a selection of entries from the schedule state $D' \subseteq \Sigma^S$:

$$D = \bigcup_{(I,\_,\_) \in D'} I$$

$$\sigma^{S\prime} = \sigma^S / D'.$$

D is the largest set that satisfies the following constraints:

$$|D| \leqslant x$$

$$\forall d \in D', \forall d' \in \sigma^S / D' : d \succeq'_t d'.$$

The pop function selects a set of inputs $D \in \mathcal{P}(\text{Inputs})$ and removes the corresponding entries from the schedule state σ^S. This selection process involves choosing a subset of the schedule state D'. The entries in this subset are considered the largest in the schedule state according to the order $\succeq'_t$. By ensuring that $|D| \leqslant x$ is as large as possible, we assure that the entire available bandwidth of x values is utilized.

To demonstrate priority constraint values and the definition of the requirements, consider the following example:

Example 3.1.2. Consider the schedule state $\sigma^S \in \Sigma^S$ when scheduling priority constraint values with a bandwidth of 2 values every 1 time unit:

$$\sigma^S = \{(\{a\}, 1, 3), (\{b\}, 4, 5), (\{c\}, 4, 7)\}$$

We consider inputs as overdue if they were not evaluated for 2 or more time units. Additionally, consider 4 as the current absolute time. The schedule state schedules the following inputs:

1. Input stream a is scheduled with the lowest priority of 3. The stream was scheduled at the absolute time 1 and is overdue, as it was scheduled longer than 2 time units ago.

2. Input streams b is scheduled with a priority of 5. It was scheduled at the absolute time 4.

3. Input stream c is scheduled with the highest priority 7. The scheduling also happened at the absolute time 4.

The next scheduled deadline always corresponds to the minimal time allowed by the bandwidth. In this example, that is 1 time unit after the current time 4:

$$\text{nextdl}^S(\sigma^S, 4) = 5$$

The order $\succeq'_t$ with the current time $t = 4$ orders the elements in σ^S in the following way:

$$(\{a\}, 1, 3) \succeq'_4 (\{c\}, 4, 7) \succeq'_4 (\{b\}, 4, 5).$$

$(\{a\}, 1, 3)$ is prioritized against all other inputs, as it is the only stream that is overdue. The second element $(\{c\}, 4, 7)$ is prioritized over $(\{b\}, 4, 5)$ as it has a higher priority of 7 compared to the priority of 5 for the last element.

According to this order, the inputs scheduled for the next deadline are the following:

$$\text{pop}(\sigma^S, 4) = (\sigma^{S\prime}, D)$$

with

$$D' = \{(\{a\}, 1, 3), (\{c\}, 4, 7)\}$$

$$D = \{a, c\}$$

$$\sigma^{S\prime} = \{(\{b\}, 4, 5)\}.$$

D' satisfies all required conditions:

1. $|D'| = 2$, because 2 is the largest number of values allowed by the bandwidth for every cycle. There can not exist a larger set, as it is required for $|D|$ to be smaller or equal to 2.
2. The order $\succeq_t'$ orders all elements in D' before all elements in $\sigma^{S\prime} = \sigma^S / D'$.

For the scheduler, this information indicates that at the next absolute deadline 5, the monitor is expecting to run a new monitoring cycle with new values for both inputs a and c. △

3.2. Syntax

For our scheduling approach, we adapt the syntax of RTLola presented in Sect. 2.1.4 to allow the inclusion of scheduling annotations. These scheduling annotations can appear in three places inside a specification: on input streams definitions, on eval clauses of output streams and trigger definitions. Fig. 3.1 depits a railroad diagram of all the grammar rules of RTLola we have changed. We optionally allow the inclusion of the red nonterminal `sannotation` after the value type of an input stream definition, at the end of an eval clause and after a trigger message. All other rules of the grammar remain unchanged.

→ Sec. 2.1.4, p. 10

Figure Fig. 3.2 depicts the syntax of these scheduling annotations in the form of a railroad diagram. Scheduling annotations, represented by the nonterminal `sannotation` in the diagrams, always have the same syntactical form. Each scheduling annotation begins with the keyword *schedule*, introducing the subsequent constraints. After that, the individual constraints are listed, separated by the keyword *and*. A constraint, referred to as the nonterminal `constraint`, can be one of three kinds: a min constraint, a max constraint, or an in constraint. All of these first introduce the kind of constraint by the keywords *min*, *max* or *in*. *Min*- and *max*-constraints impose upper and lower bounds on the constraint value respectively. Therefore both constraints are followed by the nonterminal `cvalue`, which specifies the value to be constrained. The *in*-constraint allows to specify a range of allowed values. A range is defined by an opening bracket *[*, followed by the lower bound, two dots *..* and the upper bound. The range is then closed with a closing bracket *]*. Essentially, the in constraint serves as a convenient shortcut for including both the min and max constraints.

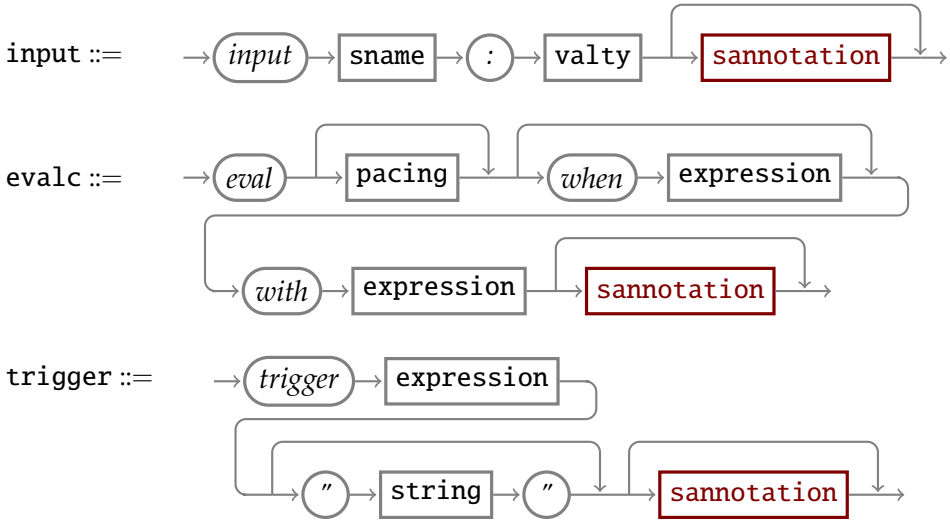


Figure 3.1.: The railroad diagram of the adapted grammar rules. The red nontermials are added to allow the inclusion of scheduling annotations on input streams, eval clauses and triggers.

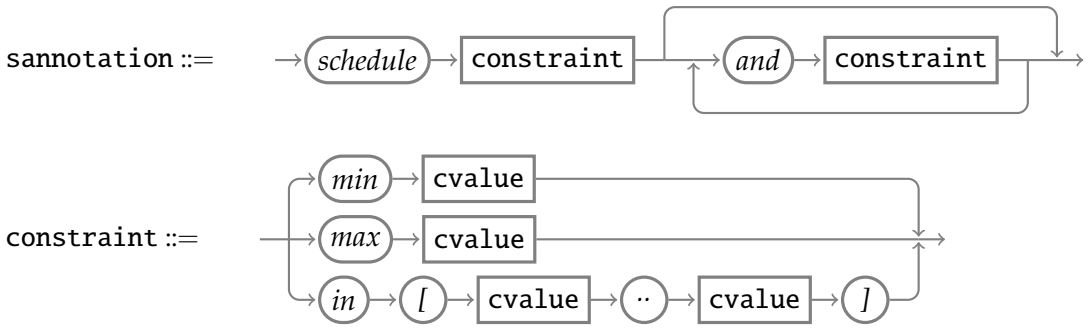


Figure 3.2.: The railroad diagram depicting the syntax of scheduling annotations. The constraints are introduced by the keyword *schedule* and separated by the keyword *and*. Each constraint can be either a *min*-, *max*- or *in*-constraint.

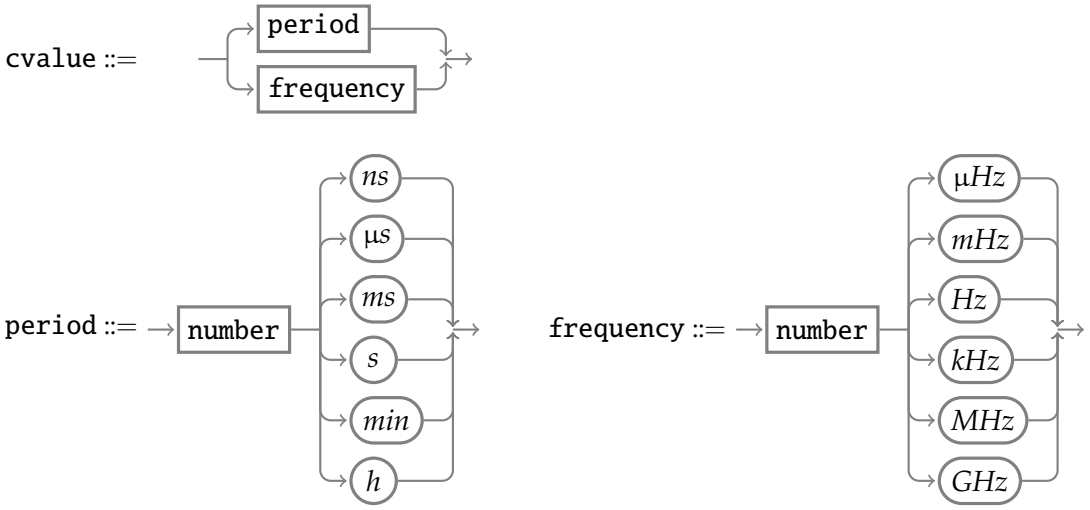


Figure 3.3.: The railroad diagram of a constraint value when scheduling over durations. The constraint value is either given as a duration directly, by a number with a time unit or as a frequency with a number and a unit in Hertz.

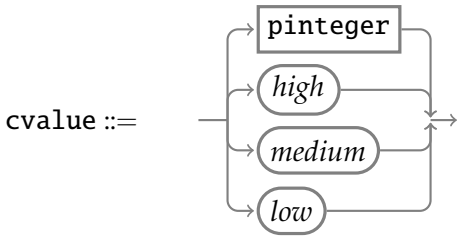


Figure 3.4.: The railroad diagram of a constraint value when scheduling over priorities. A constraint value is either one of the priority constants *high*, *medium*, or *low*, or a positive integer (represented by the `pinteger` nonterminal).

The `cvalue` nonterminal varies depending on the kind of constraint value that is used for scheduling. We will now focus on the two kinds of constraint values introduced earlier in Sect. 3.1: durations and priorites.

→ Sec. 3.1, p. 22

Fig. 3.3 depicts the railroad diagram for the `cvalue` rule, when dealing with duration constraint values. In this case, a value either represents a time period (the nonterminal `period`) or a duration (the nonterminal `duration`). A time period is given by a number followed by a time unit (e.g. `5s` or `10ms`) while a frequency is given by a number followed by a unit in Hertz (e.g. `2Hz` or `0.5Hz`). The syntax for duration constraint values corresponds to the `perty` nonterminal in regular RTLola as explained in Sect. 2.1.4 when parsing the periodic pacing type of a stream.

Fig. 3.4 depicts the syntax of the `cvalue` rule when scheduling over priority constraints. In this case, the `cvalue` can be either the positive integer directly or a constant,

which translates to such an integer. Here, the constant *low* translated to 1, *medium* to 5 and *high* to 10.

3.3. Specification Semantics

In this chapter, we specify the semantics of annotated RTLOLA specifications. First, we give an intuition about the scheduling annotations through an example. Following the example, we delve into the semantics of these scheduling annotations. Based on the annotations we construct a schedule, which maps conditions over the current monitor state to constraint values for the inputs in the specification. This schedule is utilized in the scheduled monitoring, a topic we will discuss in Sect. 3.4.

Example 3.3.1. The following example illustrates the interpretation of scheduling annotations in a specification. In this instance, we focus on scheduling *duration constraint values* (see Sect. 3.1.1). Consequently, the annotations describe the time duration after which a stream is reevaluated after it receives a new value.

```
input a : UInt64 schedule min: 1s and max: 10s
input b : UInt64

output c
    eval @(a&&b) when a ⩽ 10 with b + 1 schedule min: 3s
    eval @(a&&b) when a > 10 with b + 2 schedule min: 5s

trigger a = 0 schedule min: 5s
```

This example establishes the following constraints:

- The input stream a must receive a new value at least once every 10 seconds, but not more frequently than once per second.
- When the first eval clause of the output stream c matches, i.e. $a \leqslant 10$, then c must reevaluated after at most 3 seconds. If the second eval clause matches, i.e. $a > 10$, then it must reevaluated at most after 5 seconds.
- The trigger checking if $a = 0$ must be evaluated at least every 5 seconds.

A solution for these constraints is the following:

- If both inputs a and b received a new value, and $a \leqslant 10$, then both a and b receive a new value in 3 s.
- If both inputs a and b received a new value, but $a > 10$, then we require both a and b to receive a new value in 5 s.
- If the input a receives a new value, then we require a to receive a new value again in 5 s as well.

It is essential to note that the pacing of stream c is `@(a&&b)`, which implies that both a new value for input a and for input b is required to evaluate the output c. △

Next we formally define this intution introduced above. The analysis of the scheduling annotations will lead to a schedule function, which represents the derived schedule of the monitor:

Definition 3.3 (Schedule)

The *schedule* of an RTLola specification S maps the sets of inputs $\mathcal{P}(\text{Inputs})$ to a sorted list of filter-constraint value pairs: **Def.** Schedule

$$\text{Schedule}_S : \mathcal{P}(\text{Inputs}) \to \langle \text{Condition} \times \text{Value} \rangle$$

where Condition represents an arbitary boolean-valued RTLola expression.

The schedule function depends on which inputs have received new values in the current cycle. This dependency arises because the monitor must only reschedule inputs, that received fresh data in the current cycle. As a result, the function always plans the next evaluation of inputs based on the time of their last update. To account for dependencies within RTLola specifications, the schedule function plans sets of input streams. By doing so, it ensures that output streams like c in the example above calculate new values according to their attached constraints. If the inputs for a and b were scheduled separately, there would be no guarantee that they arrive at the same time, and therefore no guarantee for the pacing of c to allow the evaluation.

For each set of input streams, the Schedule_S function returns a sorted list of pairs. Each pair in this list consists of a condition, a boolean-valued RTLola expression, and an associated constraint value. The semantics of such a sorted list dictates that the monitor evaluates the conditions of each pair in ascending order. The value of the first condition that evaluates to true under the current monitor state is selected to determine the next time these inputs receive a new value.

Example 3.3.2. The mathematical representation of the schedule from the example above would appear as follows:

$$\text{Schedule}_S(ac) = \begin{cases} \langle (a \leqslant 10, 3\,s), (a > 10, 5\,s) \rangle & \text{if } ac = \{a, b\} \\ \langle (\top, 5\,s) \rangle & \text{if } ac = \{a\} \\ \langle \rangle & \text{otherwise} \end{cases}$$

Assume the monitor receives the event containing the new value 20 for the input a, as well as the new value 20 for the input b. As both inputs received a new value at the same time, the monitor requests the time of the next update of both inputs:

$$\text{Schedule}_S(\{a, b\}) = \langle (a \leqslant 10, 3\,s), (a \geqslant 10, 5\,s) \rangle$$

The monitor now proceeds to test the conditions in the order they are listed, based on the current state of the monitor. The first condition is found to be false: when synchronously accessing input a, it returns the newly received value 20. As this value is not lower than 10, this case is not considered. However, the subsequent condition is met: the value 20 is indeed greater than 10. Consequently, the monitor expects new values for both input streams a and b to arrive together in 5 s. △

In preparation for defining the Schedule_S function, we introduce a set of auxiliary functions. To begin, we extract all scheduling constraints within an RTLOLA specification and organize them into a set of constraints defined as follows:

$$\text{Constraints}_S : \mathcal{P}(\text{Ac} \times \text{Condition} \times \text{Constraint}).$$

Within this set, each constraint is associated with a static filter condition Ac and a semantic filter condition Condition. Both of these conditions must be met for the constraint to be considered in the scheduling process.

Remark 3.3.1. *It is important to note that scheduling annotations are only applicable for event-driven streams, given that periodic streams operate independently from any inputs and are therefore also not influenced by scheduling. Additionally, to uniquely define which inputs are required for the evaluation of a stream, we limit the event-based pacing type to only consist of conjunctions of input streams.*

Static Filter

The *static filter* $\text{Ac} = \mathcal{P}(\text{Inputs})$ corresponds to the event-driven pacing type of the stream to which the constraint is attached. Given that scheduling annotations are confined to streams with event-driven pacing types consisting of conjunctions, *we represent static filter conditions as sets of inputs*. In this representation, we deem a static filter condition fulfilled if all inputs in the set receive a new value at the same time.

Semantic Filter

In contrast, the *semantic filter* condition is represented by a boolean-valued RTLOLA expression and is derived from filter conditions of eval clauses. Constraints on input streams and triggers on the other hand always have a semantic filter condition of $\top$, as they are always evaluated whenever the static filter condition is met.

While scheduling, the two filters of a constraint signify, when the constraint has to be taken into consideration. More precisely, a constraint is only relevant if the static filter is currently active, i.e. that the current inputs to the monitor satisfy the positive boolean formula. Furthermore, the semantic filter condition must be satisfied by the current state of the monitor for the constraint to be considered.

Example 3.3.3. Consider the specification S from Example 3.3.1. For this specification, the collected constraints appear as follows:

$$\begin{aligned}\text{Constraints}_S = &\{(\{a\}, \top, \max(10\ \text{s})), (\{a\}, \top, \min(1\ \text{s}))\} && \text{for the input } a\\ &\cup \{(\{a, b\}, a \leqslant 10, \min(3\ \text{s})),\end{aligned}$$

$$\{(\{a, b\}, a > 10, \min(5\ s))\} \qquad \text{for the output } c$$
$$\cup\{(\{a\}, \top, \min(5))\} \qquad \text{for the trigger}$$

In the example, we observe one triple for each constraint in the specification. Both constraints on input stream a trivially have the static filter $\{a\}$ and the semantic filter $\top$. In the case of the output stream c, both conditions share the static filter $\{a, b\}$, which corresponds to the pacing type of stream c. The semantic filter aligns with the filter condition of the eval clause to which the constraints are attached. Regarding the constraint on the trigger, its static filter corresponds to the pacing type of the trigger, `@a`, and the semantic filter is, as for every trigger, $\top$. △

In the example above it is evident that the semantic filter of a constraint depends upon the when-conditions within the eval clauses. To extract the semantic filter condition for an eval clause, we will now introduce the filter function:

Definition 3.4 (Semantic Filter)

The *semantic filter* $\text{filter}(s, i)$ for an eval clause $i \in s.\text{clauses}$ within an output stream $s \in \text{Outputs}$, represents the condition under which the eval clause is evaluated, provided that the pacing type is satisfied:

Def. Semantic Filter

$$\text{filter} : \text{Outputs} \times \mathbb{N} \to \text{Condition}$$

$$\text{filter}(s, i) = s.\text{clause}_i.\text{filter} \wedge \bigwedge_{1 \leqslant i' < i} \neg s.\text{clause}_{i'}.\text{filter}$$

while $s.\text{clause}_i.\text{filter}$ represents the when-condition of the i'th eval clause of s if it exists and $\top$ otherwise.

Within the provided definition, the semantic filter of clause i does not simply equate to the when-condition of that particular clause. Instead, it is more accurately described as the conjunction of the eval clause's when-condition with the negation of all the when-conditions of the eval clauses that are defined before it. This is a consequence of the clause i being chosen only when none of the preceding clauses match.

Example 3.3.4. Consider the output stream c from Example 3.3.1:

```
output c
    eval when a ⩽ 10 with b + 1 schedule min: 3s
    eval when a > 10 with b + 2 schedule min: 5s
```

The filter condition of the second eval clause $\text{filter}(c, 2)$ is

$$\text{filter}(c, 2) = a > 10 \wedge \neg(a \leqslant 10) = a > 10.$$

This occurs because when the second eval clause is evaluated, not only must the filter condition of that second clause evaluate to true, but also the filter condition of the first clause must evaluate to false. This is necessary because otherwise, the first clause would have been evaluated instead, as the conditions are tested from top to bottom. △

Using the filter function to extract the semantic filter of an eval clause, we can now define the set of constraints for an annotated specification:

Definition 3.5 (Constraints)

Def. Constraints

The *constraints of the specification* S consisting of inputs I, event-based outputs O and triggers T, are at a set of triples. Each triple corresponds to an individual constraint in the specification. In these triples, the first element signifies the static filter, the second element represents the semantic filter and the third element is the constraint itself:

$$\text{Constraints}_S : \mathcal{P}(\text{Ac} \times \text{Condition} \times \text{Constraint})$$

$$\begin{aligned}\text{Constraints}_S = &\{(ac, \top, c) \mid \exists i \in I : ac = \{i\} \wedge c \in i.\text{constraints}\} \\ &\cup \{(ac, f, c) \mid \exists o \in O : ac = o.\text{pacing} \\ &\quad \wedge \exists c \in o.\text{clauses} : f = \text{filter}(o, c) \wedge c \in c.\text{constraints}\} \\ &\cup \{(ac, \top, c) \mid \exists t \in T : ac = t.\text{pacing} \wedge c \in t.\text{constraints}\}\end{aligned}$$

where o.clauses denotes the set of all eval clauses associated with output stream o, and s.constraints refers to the set of constraints annotated to the input stream, eval clause or trigger s. Additionally, s.pacing refers to the set representation of the pacing type of the output stream or trigger s.

The constraints of a specification consist of three parts: the constraints associated with input streams, constraints associated with eval clauses and constraints on triggers.

For inputs, the static filter is straightforward. It is consistently formed by the input itself. In other words, the input must be active for the constraint to be considered. In terms of the semantic filter, for inputs, it is always $\top$.

Moving on to eval clauses, it becomes slightly more intricate. Here, the static filter is determined by the event-based pacing of the corresponding output stream. The semantic filter for eval clauses is determined with the filter function introduced above.

Similar to output streams, triggers have a static filter that corresponds to their event-based pacing types. However, as triggers can not have when-conditions, the semantic filter for triggers is uniformly $\top$.

With this understanding of constraints, we now define a function to access whether a triple Ac $\times$ Condition $\times$ Value is compatible with all constraints collected from a specification:

Definition 3.6 (Satisfies)

The function call satisfies$(S, (ac, f, v))$ returns true, if the triple $(ac, f, v) \in \text{Ac} \times \text{Condition} \times \text{Value}$ *satisfies* all the constraints inside the set of constraints S: **Def.** Satisfies

$$\text{satisfies} : \text{Constraints} \times (\text{Ac} \times \text{Condition} \times \text{Value}) \to \mathbb{B}$$

$$\text{satisfies}(C, (ac, f, v)) = \forall (ac', f', cs) \in C :$$

$$(ac \Rightarrow ac' \wedge f' \Rightarrow f) \Rightarrow \begin{cases} v' \leqslant v & \text{if } cs = min(v') \\ v' \geqslant v & \text{if } cs = max(v') \\ v' \leqslant v \leqslant v'' & \text{if } cs = in(v', v'') \end{cases}$$

To assess whether a triple (ac, f, v) complies with all constraints in a set C, it is necessary to examine all constraints $(ac', f', cs) \in C$ that are implied by the given triple. Specifically, this implies that the triple must only satisfy those constraints, that are implied by the static and semantic filter of the triple. Therefore, we first verify if $ac \Rightarrow ac'$ and $f' \Rightarrow f$. Only when both these implications hold do we proceed to determine whether the constraint value v aligns with the constraint itself.

if the constraint is of the "min" type, v must be greater than or equal to the given bound. For "max" constraints, it must be less than or equal. In the case of a range constraint, it naturally must fall between the lower and upper bounds.

Example 3.3.5. Assume we extracted the two constraints

$$C = \{(\{a\}, \top, min(5)), (\{a, b\}, a > 10, max(10))\},$$

from an RTLola specification. We now assess whether the constraints in C allow the inclusion of the triple $(\{a, b\}, a < 5, 12)$ in our schedule, i.e. whether

$$\text{satisfies}(C, (\{a, b\}, a < 5, 12))$$

is true.

First, we need to identify all constraints in C that are relevant for that triple. The first constraint in C originates from a stream with a static filter of a. We must consider this constraint because $a \wedge b \Rightarrow a$, i.e. every time the streams a and b receive a new value, also a stream with the pacing type a receives a new value. Additionally, every condition is implied by the semantic filter $\top$. As $12 > 5$, the constraint value 12 satisfies the first constraint $min(5)$.

The second constraint in C does not have to be considered. While the pacing types are equal, the semantic filters do not imply each other.

As the tuple satisfies all relevant constraints, it is permissible for a schedule to include the triple $(\{a, b\}, a < 5, 12)$. △

With these auxiliary functions, we now have all the building blocks to calculate a schedule for an annotated RTLOLA specification:

Definition 3.7 (Schedule)

Def. Schedule

The *schedule of an annotated RTLOLA specification* S for a set of inputs $ac \in Ac$, with collected constraints $C = \text{Constraints}_S$, is the list $\langle f_i, v_i \rangle_{i \leqslant n}$ with

$$\text{Schedule}_S : Ac \to \mathcal{P}(\langle \text{Condition} \times \text{Value} \rangle)$$

$$\text{Schedule}_S(ac) = \langle f_i, v_i \rangle_{i \leqslant n}$$

$$\begin{aligned} \text{with} \quad & \text{satisfies}(C, (ac, f_i, v_i)) \\ & \wedge \neg \exists v' : v_i \preceq v' \wedge \text{satisfies}(C, (ac, f_i, v')) \\ & \wedge \forall i' < i : v_i \preceq v_{i'}. \end{aligned}$$

For a schedule to be deemed valid, it must adhere to the following constraints:

1. Each element within the schedule for a given set of inputs ac must satisfy the constraints collected from the annotated specification S.

2. There must not exist a greater (according to the total order of the constraint value, see Def. 3.2) constraint value v' that could be scheduled instead while still satisfying all collected constraints. In essence, this means that the schedule always prioritizes the best available value.

3. All constraint values preceding the current entry are greater than the current value. This implies that conditions resulting in greater constraint values are tested first. Only when these conditions fail to produce a match, larger constraint values are considered. Because of this constraint, the constraint values in a schedule are consistently sorted in descending order.

   Remark 3.3.2. *This is according to the total order of the constraint value. For durations, for example, the schedule is sorted from shortest to longest duration, as shorter durations are considered greater according to the total order.*

3.4. Monitor Semantics

In Sect. 2.1.5, we introduced the monitor semantics for regular RTLOLA. In the following section, we extend this monitor semantics to accommodate our scheduling approach.

→ Def. 2.1, p. 14

Let's revisit the definition of the regular RTLOLA semantics as presented in Def. 2.1.

The semantics is given by the asynchronous state-based time-driven monitor function M:

$$M : \left\langle \mathcal{T}_i^{\downarrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\downarrow}} \times \Sigma^M \times \mathbb{R}^+ \to \left\langle \mathcal{T}_i^{\uparrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\uparrow}} \times \Sigma^M \times \mathbb{R}^+$$

The function M operates by processing a single event, which consists of a set of input values at a specific time. This function executes the monitor with the new input values and computes new values for output streams and triggers. It also receives an internal state of the current monitor which contains previous stream values. As the monitor evaluates the new event, it also updates the internal state accordingly. The state returned by processing an event is then utilized in the next function call to process the next event. This allows each event to modify the internal state of the monitor.

In this section, we will adapt this definition for the scheduled monitoring approach:

Definition 3.8 (Scheduled Monitor Function)

Given a schedule S, a set of new input values $\left\langle \mathcal{T}_i^{\downarrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\downarrow}}$, an internal state $\Sigma^{M,S}$ and a time $\mathbb{R}^+$, the *scheduled asynchronous state-based time-driven monitor function* M^S calculates one cycle of the scheduled monitor:

Def. Scheduled Monitor Function

$$M^S : \left\langle \mathcal{T}_i^{\downarrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\downarrow}} \times \Sigma^{M,S} \times \mathbb{R}^+ \to \left\langle \mathcal{T}_i^{\uparrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\uparrow}} \times \Sigma^{M,S} \times \mathbb{R}^+ \times \mathcal{P}(\text{Inputs})$$

The internal state $\Sigma^{M,S}$ consists of a pair (Σ^M, Σ^S), with Σ^M representing the state of the monitor already used in the regular RTLola semantics, while Σ^S with $\Sigma^S = \mathcal{P}(\mathcal{P}(\text{Inputs}) \times \mathbb{R}^+ \times \text{Value})$ representing the state of the schedule.

The semantics of scheduled RTLola differs from the original semantic in two aspects: 1. A different internal state, and 2. an additional element in the returned tuple. The internal state now consists of two components: the state of the monitor, which contains the previous values of streams, and the current schedule state, maintaining the scheduled constraint values for the inputs. This schedule state, introduced in Def. 3.1, is represented as a set of triples, each mapping sets of input streams to constraint values together with the time of the scheduling.

→ Def. 3.1, p. 21

As inputs are not scheduled in regular RTLola, new input values correspond to a new call to the monitor whenever they arrive. Additionally, the monitor returns a time when an, maybe empty, update to the monitor is required to calculate new values of periodic streams. The scheduled version not only returns this time but also a set of input streams that are due at that time. This means that the monitor requires the given inputs to receive new values with the next call to the monitor at that given time. In the case where the next deadline of a periodic stream is earlier than any scheduled input streams, the list of inputs due at that time is empty.

We demonstrate the scheduled RTLola semantics with the following example:

Example 3.4.1. Assume we have an RTLola specification with the input streams a and b and derive the following schedule based on the duration scheduling annotations inside the specification:

$$S = \big\{(\{a\}, \langle (a = 0, 2), (a \neq 0, 3)\rangle),$$
$$(\{b\}, \langle (\top, 3)\rangle)\big\}$$

This schedule specifies the following requirements: if the monitor receives a value for the input stream a and at that time the condition $a = 0$ holds, the monitor requires a new value for a in 2 time units. If $a \neq 0$ holds when a receives a new value, the monitor requires a new value for a only in 3 time units. Additionally, the input stream b should receive a new value every 3 time units, regardless of the current state of the monitor.

Initially, all inputs automatically receive new values when the monitoring starts. In this example, we assume that initially, at the start time of 1, both a and b have a value of zero. Additionally, the internal state, i.e. both all the output streams as well as the schedule, is empty initially. It therefore consists of Σ_0^M, the initial monitor state for regular RTLola semantics (see Sect. 2.1.5) and an empty set for the schedule.

Giving these inputs to the monitor to compute the first cycle, we receive the following output:

$$M^S(\langle 0, 0\rangle, (\Sigma_0^M, \emptyset), 1) = (V, (\Sigma_1^M, \{(\{b\}, 1, 3)\}), 3, \{a\})$$

The monitor function M^S produces a verdict V, the new value of all the outputs and triggers that received a new value in this cycle of the monitor. Furthermore, it produces a new monitoring state Σ_1^M, with updated information on the output streams based on the new inputs. From the scheduler state $\{(\{b\}, 1, 3)\}$ together with the next deadline at $3, \{a\}$ it is evident that both a and b have been scheduled during this cycle. Specifically, a is scheduled to receive a new value at time 3, while b is set to receive a new value at time $1 + 3 = 4$. The reason for this is, that based on the inputs $\langle 0, 0\rangle$, the condition $a \neq 0$ holds, which therefore schedules the new value for a at time 1 + 2 = 3. Likewise, given that $\top$ always matches, the next value for b is scheduled in 3 time units. Given that the monitor returned the time 3 as the next deadline, the subsequent call to the monitor should provide the new value of the input a at time 3.

The next step is to wait until that deadline is reached and then query for a new value of the input stream a. For this example, assume that the new value for a at time 3 is 1. As the input b was not scheduled for time 3, b will not receive a new value. We compute the next cycle of the monitor by calling M^S with the new input value and the states returned by the last monitoring cycle:

$$M^S(\langle 1, \bot\rangle, \Sigma_1^M, \{(\{b\}, 1, 3)\}, 3) = (V', (\Sigma_2^M, \{(\{a\}, 3, 3)\}), 4, \{b\})$$

Once again, the monitor computes an updated state and returns the verdict based on that new state. In this instance, the next cycle of the monitor is scheduled to run at time 4 with a new value for the input stream b. Input stream a does not receive a new value at that time, but rather at time $3 + 3 = 6$. △

The following definition presents the adapted monitor function introduced above formally:

Definition 3.9 (Scheduled Monitor Function)

Given an event $ev \in \left\langle \mathcal{T}_i^{\downarrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\downarrow}}$, a monitor state $\sigma \in \Sigma^{M,S}$, a timestamp $t \in \mathbb{R}^+$ and

$$\sigma' = \text{update}(\sigma, ev, t)$$
$$(\sigma'', t', S) = \text{nextdl}(\sigma', t),$$

we define the *scheduled asynchronous state-based real-time monitor function* as

$$M^S : \left\langle \mathcal{T}_i^{\downarrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\downarrow}} \times \Sigma^{M,S} \times \mathbb{R}^+ \to \left\langle \mathcal{T}_i^{\uparrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\uparrow}} \times \Sigma^{M,S} \times \mathbb{R}^+ \times \mathcal{P}(\text{Inputs})$$
$$M^S(ev, \sigma, t) = (\text{slice}(\sigma''), \sigma'', t', S)$$

The definition closely follows the definition of the monitor function for original RTLola shown in Def. 2.5. Different however are the slice, nextdl and update functions, which we have adapted to take the schedule into account. slice now operates on the new internal state, the tuple of monitor state and schedule and the update function additionally to the monitor also updates the schedule. The nextdl function is capable of modifying the internal state, as it might remove old elements from the schedule. Additionally, the nextdl function returns a set of inputs that are due at the given deadline. All of these functions still internally use the same-named functions from the original RTLola semantics. As the original functions are defined on the monitor state, we can use them by referring to the first element of the internal state. In the following, we will refer to these original functions as slice^M, nextdl^M and update^M respectively.

→ Def. 2.5, p. 17

For the *slice function*, we simply adapt to the changed internal state. Since the state $\Sigma^{M,S} = (\Sigma^M, \Sigma^S)$ now consists of a tuple of monitor state and schedule, the slice function simply calls the original slice function slice^M on the first tuple element:

Def. Slice

$$\text{slice} : \Sigma^{M,S} \to \left\langle \mathcal{T}_i^{\uparrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\uparrow}}$$
$$\text{slice}((\sigma^M, \sigma^S)) = \text{slice}^M(\sigma^M).$$

The nextdl function has to take two possible deadlines into account. First, the deadline for the next periodic stream - the original nextdl^M function. Additionally, the schedule

also provides a next deadline: The nextdl^S function provided by the used constraint value and introduced in Def. 3.2. This function examines the schedule Σ^S and the current time $\mathbb{R}^+$ and returns the time, when the next scheduled inputs are due. The overall nextdl function checks both of these deadlines and returns the earlier one:

→ Def. 3.2, p. 22

Definition 3.10 (Next Deadline)

Def. Next Deadline

The *next deadline* $\text{nextdl}(\sigma, t)$ of a state $\sigma = (\sigma^M, \sigma^S) \in \Sigma^{M,S}$ and the current time $t \in \mathbb{R}^+$ is a tuple $\mathbb{R}^+ \times \Sigma^{M,S} \times \mathcal{P}(\text{Inputs})$ consisting of the next deadline of the monitor with the inputs scheduled for that time and a possibly updated internal state:

$$\text{nextdl} : \Sigma^{M,S} \times \mathbb{R}^+ \to \mathbb{R}^+ \times \Sigma^{M,S} \times \mathcal{P}(\text{Inputs})$$

$$\text{nextdl}(\sigma, t) = \begin{cases} (\text{nextdl}^M(\sigma^M), \sigma, \emptyset) & \text{if } \text{nextdl}^M(\sigma^M) < \text{nextdl}^S(\sigma^S, t) \\ (\text{nextdl}^S(\sigma^S, t), (\sigma^M, \sigma^{S\prime}), I) & \text{if } \text{nextdl}^M(\sigma^M) \geqslant \text{nextdl}^S(\sigma^S, t) \end{cases}$$

where

$$(\sigma^{S\prime}, I) = \text{pop}(\sigma^S, t)$$

This function checks which deadline is earlier, the next periodic deadline or the next scheduled deadline. If the next periodic deadline is earlier (first case), the function simply returns the time of that deadline. Since a periodic output stream is due at that time, no inputs are required. Therefore the set of inputs is empty in this case. Additionally, the internal state remains the same as the schedule does not change when updating periodic output streams.

If the next scheduled deadline is earlier (second case), the function also returns the time of that deadline. Additionally, the internal state is modified by a call to the pop function provided by the constraint value and introduced in Def. 3.2. This function removes all the inputs that are scheduled at the next scheduled deadline and returns them as a set.

→ Def. 3.2, p. 22

The update function has two tasks in the scheduled semantics: First to update the monitor state based on the new inputs and second to update the schedule according to the new monitor state:

Definition 3.11 (Update Function)

Def. Update Function

Given an internal state $(\sigma^M, \sigma^S) \in \Sigma^{M,S}$, an event $ev \in \left\langle \mathcal{T}_i^{\downarrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\downarrow}}$ and a timestamp $t \in \mathbb{R}^+$, the *update function* updates the internal state according to that event:

$$\text{update} : \Sigma^{M,S} \times \left\langle \mathcal{T}_i^{\downarrow} \cup \{\bot\} \right\rangle_{i \leqslant n^{\downarrow}} \times \mathbb{R}^+ \to \Sigma^{M,S}$$

$$\text{update}((\sigma^M, \sigma^S), ev, t) = (\sigma^{M\prime}, \text{update}^S(\sigma^S, \sigma^{M\prime}, t))$$

with

$$\sigma^{M\prime} = \text{update}^{M}(\sigma^{M}, e\nu, t).$$

To update the monitor state, the update function utilizes the update^{M} function from regular RTLola semantics (see Def. 2.4). Analogous to the update^{M} function, we now define a update^{S} function to update the schedule based on the current monitor state. This function makes use of a schedule S, which is derived from the scheduling annotations inside the specification and is precomputed before monitoring starts. This step is covered in Sect. 3.3.

→ Sec. 3.3, p. 32

Definition 3.12 (Scheduled Update Function)

Given a schedule S, a monitor state $\sigma^{M} \in \Sigma^{M}$ and a schedule $\sigma^{S} \in \Sigma^{S}$, the state $\text{update}^{S}(\sigma^{S}, \sigma^{M}) \in \Sigma^{S}$ is the *updated schedule state* taking the new monitor state into account:

Def. Scheduled Update Function

$$\begin{aligned}
&\text{update}^{S} : \Sigma^{S} \times \Sigma^{M} \times \mathbb{R}^{+} \to \Sigma^{S} \\
&\text{update}^{S}(\sigma^{S}, \sigma^{M}, t) = \sigma^{S} \cup \big\{(ac, t, v) \mid ac \in \mathcal{P}(\text{Inputs}) \\
&\qquad\qquad \wedge\, \text{isfresh}(ac, \sigma^{M}) \\
&\qquad\qquad \wedge\, \exists i \in \mathbb{N} : v = \text{value}(S, ac, i) \\
&\qquad\qquad \wedge\, [\![\text{cond}(S, ac, i)]\!]_{\sigma^{M}} \\
&\qquad\qquad \wedge\, \forall i' < i : \neg\, [\![\text{cond}(S, ac, i')]\!]_{\sigma^{M}} \big\}
\end{aligned}$$

while $[\![e]\!]_{\sigma^{M}}$ evaluates the RTLola expression e under the monitor state σ^{M} and $\text{isfresh}(ac, \sigma^{M})$ checks if all inputs in ac received a new value in the last cycle. $\text{cond}(S, ac, i) = S[ac][i][0]$ is the semantic filter condition of the i'th constraint value in the schedule S for static filter ac and $\text{value}(S, ac, i) = S[ac][i][1]$ likewise is the corresponding constraint value.

The monitor exclusively reschedules inputs that have received a new value during the current cycle. Therefore, the update function exclusively examines inputs, that are currently considered fresh, meaning they have received a new value during the current cycle. For each input with scheduling information, the update function scans the schedule for the first condition that evaluates to true under the current monitor state. The value associated with this condition is then included in the schedule state for that particular set of inputs. Furthermore, to ensure that the first matching condition is selected, all preceding conditions in the sorted list must evaluate to false under the given monitor state.

3.5. Analysis

We designed our scheduling approach to allow the analysis of annotated RTLola specifications to align with the analysis of regular RTLola introduced in Sect. 2.1.6. In this section, we explore the implications of introducing scheduling on the dependency graph, memory bounds and the analysis of types in specifications.

→ Sec. 2.1.6, p. 17

3.5.1. Dependency Graph

The dependency graph of scheduled RTLola corresponds completely to the dependency graph of regular RTLola. The introduction of scheduling annotations just adds additional information to individual streams but does not change or add any dependencies to other streams. Consequently, the overall dependency graph, as well as the evaluation order, remains unaffected by the presence of scheduling annotations.

3.5.2. Memory Bound

The memory bound of a specification quantifies the memory required for monitoring a specification. It involves calculating the number of values that streams and windows need to store. Importantly, the inclusion of scheduling annotations does not impact the memory bound of any streams in the specification. However, storing information related to scheduled inputs does introduce an additional memory requirement. Notably, this additional memory is bounded by the number of event-based pacing types present in the specification. For our subset of the RTLola language, this implies that the overall amount of memory for scheduled monitoring remains bounded, an important consideration for ensuring reliable monitoring of cyber-physical systems.

3.5.3. Type System

The type analysis of RTLola remains consistent even with the introduction of scheduling annotations. Specifically, given that the dependency graph is utilized for deriving pacing types of streams, and as established earlier, the dependency graph remains unchanged, the derived pacing types also remain unchanged.

Despite the introduction of scheduling annotations, the derived pacing types remain correct as we have designed the scheduler to consider the pacing types throughout the scheduling process. By taking the pacing types into account, the scheduler schedules all necessary inputs for the evaluation of an output stream. Therefore, the scheduler ensures that no stream remains unevaluated simply due to inputs not arriving in a manner that satisfies the pacing type of the stream.

3.6. Algorithms

In this section, we explore the algorithms required to build a schedule based on scheduling annotations inside an annotated specification.

3.6.1. Collecting Constraints

The first step of translating RTLola specifications into a schedule involves extracting all annotations from the specification. This step is analogous to the semantics defined in Def. 3.5. The pseudocode depicted in Alg. 1 displays the algorithm, taking an RTLola specification as input and returning a set of constraints. Each constraint is annotated with both a static and semantic filter condition.

→ Def. 3.5, p. 36

The algorithm consists of three segments: 1. it collects all annotations associated with input streams, 2. it processes constraints related to output streams, and 3. it handles constraints associated with triggers. All these segments push constraints to a set C, which is the final output of the algorithm.

For input streams, the algorithm iterates through all inputs, and for all inputs with attached annotations pushes the constraints into the set C. Each constraint's static filter solely consists of the corresponding input i (Line 5), and its semantic filter is always $\top$ (Line 6) as specified in the semantics definition.

For output streams, the algorithm iterates through all outputs and gathers all constraints associated with eval clauses of these outputs. Notably, if an eval clause possesses scheduling annotations, it indicates that the output stream is event-driven, as explained in Sect. 3.3. The algorithm issues an error, if an annotation is present on a time-driven output stream (Line 14). As elucidated in Sect. 3.3, semantic filter conditions of constraints are constructed as the when-condition of the corresponding eval clauses and the negation of the when-conditions of all previous clauses. For this, the algorithm introduces a variable f, initialized with $\top$ before handling the first eval clause (Line 10). It subsequently appends the negation of the when-condition of eval clauses already processed by the algorithm (Line 20). The semantic filter of a constraint now comprises the negation of all proceeding when-conditions contained in the variable f together with the when-condition of the associated eval clause (Line 17). The static filter of the output stream's constraint aligns with the event-based pacing type of that stream (Line 16).

Lastly, the algorithm collects all constraints related to triggers. In this context, the static filter corresponds to the event-based pacing type of the trigger, while the semantic filter is consistently set to $\top$. After appending all constraints to the set C, the set is returned by the algorithm. This resulting set is utilized when computing the schedule for specifications.

Algorithm 1: Collect constraints from RTLola specification

Input: An RTLola specification S
Output: A set of constraints C

```
1  C ← empty set
   // collect annotations on inputs
2  for input i in S do
3      if i has scheduling annotations then
4          for every annotation c on i do
5              s ← {i}                                          // static filter
6              f ← ⊤                                            // semantic filter
7              c ← new constraint(c, s, f)
8              append c to C

   // collect annotations on eval clauses
9  for output o in S do
10     f ← ⊤                    // holds negation of previous eval clauses
11     for eval clause e in o do
12         when ← filter condition of e (or ⊤ if missing)
13         if e has scheduling annotations then
14             assert o is event-driven
15             for every annotation c on e do
16                 s ← (event-based) pacing type of e           // static filter
17                 f′ ← f ∧ when                                // semantic filter
18                 c ← new constraint(c, s, f′)
19                 append c to C
20         f ← f ∧ ¬when            // add negation of when condition to f

   // collect annotations on triggers
21 for trigger t in S do
22     if t has scheduling annotations then
23         assert t is event-driven
24         for every annotation c on t do
25             s ← (event-based) pacing type of t               // static filter
26             f ← ⊤                                            // semantic filter
27             c ← new constraint(c, s, f)
28             append c to C
29 return C
```

3.6.2. Constructing the Schedule

The construction of a schedule from a set of constraints is a two-step process: First, we gather all constraints that contribute to a specific static filter condition and compute schedules for these sets of constraints individually. The final schedule for the entire specification is then constructed by combining the schedules obtained for the different static filter conditions.

Algorithm 2: Splitting constraints into static filter conditions

Input: A set of constraints C
Output: A mapping M of static filter conditions to sets of constraints

```
// I maps static filter c to sets of static filter c' with c ⇒ c'
I ← empty map
for every static filter condition s of constraints in C do
    for every static filter condition s' of constraints in C do
        if s ⇒ s' then
            append s' to I[s]
M ← empty map
for constraint c ∈ C with static filter s do
    for static filter s' ∈ I[s] do
        append c to M[c']
return M
```

We assign each constraint to the static filter condition it belongs to. However, as seen in Def. 3.7, a constraint does not only contribute to the schedule of its static filter but also all schedules with static filters implying its static filter. Alg. 2 depicts the algorithm to produce this assignment. The algorithm returns a map, mapping static filter conditions to sets of constraints, which all are active under the given static filter condition.

First, the algorithm establishes a map of static filter conditions I, mapping each condition to all conditions implied by it. To achieve this, the algorithm iterates over all pairs (s, s') of static filter conditions and appends s' to the mapping of s, if s implies s'. This map is utilized by the subsequent phase of the algorithm: Here, the algorithm iterates over all constraints and assigns them to the corresponding static filter in the map M. Importantly, the algorithm assigns them to all static filters computed in the first step and stored in the map I for the static filter of the given constraint. Once all constraints are assigned, the algorithm returns the resulting map M.

Upon splitting the constraints based on static filter conditions, the next step involves computing a schedule for all constraints associated with a single static filter condition. The algorithm outlined in Alg. 3 describes this progress. It receives all constraints

Algorithm 3: Building schedule for individual static filter

Input: A set of constraints C with static filter s
Output: A schedule for static filter s

```
// Split constraints into filtered and total constraints
total ← empty set
filtered ← empty list
for c ∈ C do
    if semantic filter of c is ⊤ then
        append c to total
    else
        append c to filtered

// Build schedule from filtered constraints
schedule ← empty list
for c ∈ filtered do
    // Max-constraints are not allowed to contain semantic filter
    assert c defines lower bound
    v ← lower bound of c
    cond ← semantic filter of c
    append (cond, v) to schedule
Sort(schedule) descending according to total order
// Compute bounds of total constraints
lb ← greatest lower bound of constraints in total
ub ← smallest upper bound of constraints in total

// Add fallback case
append (⊤, ub) to schedule

// Schedule has to satisfy bounds of total constraints
for constraint value v in schedule do
    assert lb ⩽ v ⩽ up
return schedule
```

associated with a single static filter as an input and produces a schedule based on these constraints, a list of condition-constraint value pairs, as an output.

Starting with the algorithm, the constraints are divided into two distinct sets:

1. In the set `total` (Line 5): Constraints reasoning over all monitoring states, i.e. constraints where the semantic filter condition is $\top$.
2. In the set `filtered` (Line 7): Constraints reasoning over specific monitoring states, i.e. constraints with a semantic filter condition other than $\top$.

The algorithm uses the latter set to construct the schedule. Meanwhile, the former set is used to check the computed schedule for validity.

The algorithm directly derives the schedule from the constraints in the set of filtered constraints. To achieve this, each constraint in the set has to provide a lower bound on the constraint value, effectively being a min-constraint. Subsequently, for each constraint, the algorithm selects the lowest possible value (Line 11) and organizes them in descending order (Line 14).

Remark 3.6.1. *The presented version of the algorithm introduces a restriction on scheduling annotations; specifically, disallowing maximum constraints to have a semantic filter condition other than $\top$. This requirement arises because the algorithm only takes the lower bounds of filtered constraints into account.*

The resulting list already serves as the schedule. However, the algorithm incorporates a fallback case for the scenario where none of the semantic filter conditions match. This fallback case corresponds to the smallest upper bound among all total constraints and is appended to the end of the schedule with a semantic filter of $\top$ (Line 17).

The final step involves verifying if the schedule satisfies the bounds induced by the `total` constraints. If not every constraint value in the schedule satisfies these bounds, the specification is rejected. In this case, the algorithm is not able to construct a schedule for the given set of scheduling annotations. If all checks are satisfied, the algorithm returns the constructed schedule. Iterating this process for all static filters computed in the preceding step yields the overall schedule for the whole specification.

Leveraging the preceding algorithms, we are now able to convert an annotated RTLola specification into a schedule. The algorithm outlined in Alg. 4 encapsulates this process. Initially, the algorithm extracts all constraints from the specification by employing Alg. 1. Subsequently, the obtained set of constraints is split based on static filter conditions through the use of Alg. 2. For each distinct static filter condition, an individual schedule is then constructed using Alg. 3 and combined into a unified schedule for the entire specification.

Remark 3.6.2. *Note that the presented algorithm to compute a schedule based on a set of scheduling constraints exhibits potential for enhancements in future work. Exploring a more sophisticated scheduling algorithm, such as one leveraging a constraint solver, could allow*

Algorithm 4: Compute schedule from annotated RTLola specification

Input: An annotated RTLola specification S
Output: A schedule

```
1 C ← collect constraints for S with Alg. 1
2 M ← split constraints C with Alg. 2
3 schedule ← empty set
4 for static filter s mapping to set of constraints C in M do
5 |   schedule' ← compute schedule for C with Alg. 3
6 |   append (s, schedule') to schedule
7 return schedule
```

more advanced scheduling capabilities and address the current limitations on disallowing max constraints with a semantic filter condition other than $\top$.

Chapter 4

Implementation

This chapter explores the implementation details of the theoretical concepts introduced in the previous chapter.

Fig. 4.1 depicts a diagram showing a general overview of the monitoring approach. The input specification (1) is depicted on the left side of the diagram. This RTLola specification contains scheduling annotations, utilizing the syntax introduced in Sect. 3.2 and depicted as little red boxes in the diagram. A translator (2) analyzes the annotated specification, generating a regular RTLola specification as an output. This regular RTLola specification consists of two sections: Firstly, it encompasses all streams and triggers from the input specification while removing any constraints (3). Additionally, the constraints are analyzed by the translator and integrated as additional streams (4) into the resulting specification.

We constructed the scheduling approach in a way that there is no need to re-implement the RTLola monitor itself but take advantage of the already existing RTLola backends. Therefore, during monitoring, the resulting specification is executed on an RTLola backend (5). However, the monitor does not receive new inputs directly from sensors; instead, it interfaces with sensors through an intermediary scheduler (6). This scheduler utilizes the information provided by the additional streams (4) to determine when new inputs are required. Subsequently, the scheduler queries the sensors (7) for new values at the corresponding times and passes these values to the underlying monitor backend.

In the following sections, we will explore the different parts of the approach depicted above. The first few sections will focus on the translation progress, while Sect. 4.3 deals with the scheduling component during runtime.

The translation progress on the other hand consists of multiple steps, illustrated in Fig. 4.2. There already exists an RTLola frontend capable of parsing and analyzing RTLola specifications. We have adapted the frontend to accommodate scheduling

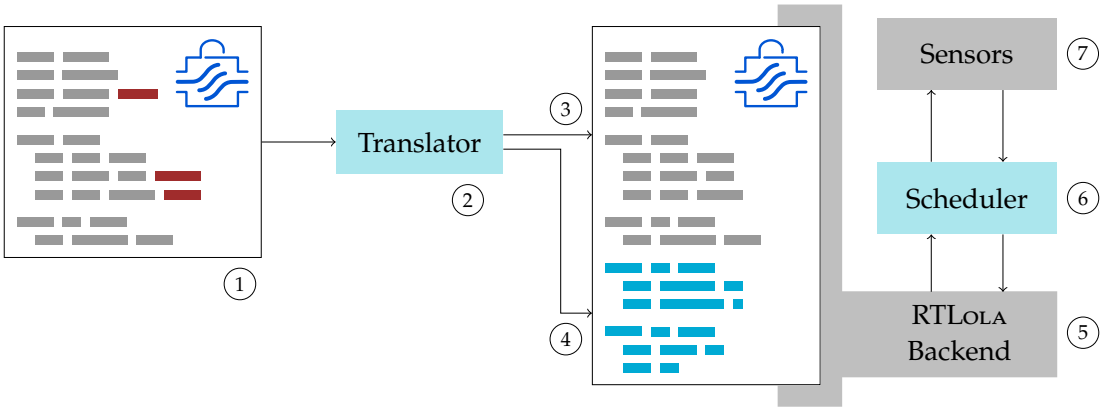


Figure 4.1.: Diagram depicting an overview of the scheduling approach. An annotated specification (1) is translated into a regular RTLola specification, consisting of the original specification without any annotations (3) and additional streams (4), containing the computed schedule. The resulting specification is then monitored with an RTLola backend (5), while a scheduler (6) sitting between monitor and sensors is responsible to query inputs from sensors (7) according to the information provided by the streams containing the schedule.

annotations and use it as the first step in the translation progress (2). The adaption of the RTLola frontend is presented in Sect. 4.1.

The next step involves the implementation of a translator, elaborated in Sect. 4.2, which utilizes the results from the modified frontend to analyze and translate the scheduling annotations into a conventional RTLola specification. In this step, constraints are extracted (3) from the remainder of the specification and analyzed (4) to construct a schedule. The resulting schedule is incorporated into the regular RTLola specification as additional streams (5), complementing the rest of the specification without any constraints (6).

The entire implementation process was executed utilizing the Rust programming language, leveraging its features to construct a safe and efficient implementation and seamlessly integrate within the existing RTLola framework.

4.1. Frontend

The RTLola frontend [4] is responsible to parse and analyse an RTLola specification and subsequently representing it in an intermediate representation (MIR). This MIR includes all streams and triggers from the specification, along with the outcomes of the analysis, such as dependencies, inferred types, and memory bounds. In the context of scheduled RTLola, as discussed in Sect. 3.5, the analysis process does not differ from that of regular

→ Sec. 3.5, p. 44

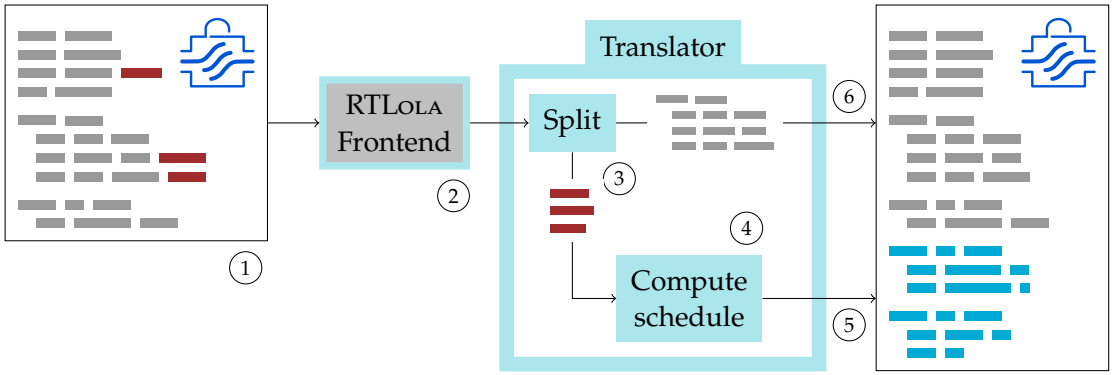


Figure 4.2.: Diagram depicting an overview of the translation progress. The annotated specification (1) is parsed and analyzed by the modified RTLola frontend (2). The resulting representation is split (3) into constraints and streams and triggers without any constraints. The latter is passed on to the resulting specification (6). The constraints are analyzed (4) to compute a schedule, which is represented as additional streams (5) in the resulting specification.

RTLola while ignoring all scheduling annotations. The scheduled approach therefore only requires small modifications of the RTLola frontend. Specifically, we extend the parser to accept scheduling annotations inside specifications. During the analysis phase, we treat scheduled specifications the same as regular ones and disregard the scheduling annotations for the analysis itself. Instead, we extend the MIR to include the scheduling annotations as additional information on streams and triggers.

To achieve the required modifications, we adapt two different stages of the RTLola frontend: the parser and the intermediate representation.

4.1.1. Parsing

The RTLola frontend employs the pest [5] parser generator to parse specifications. As a parser generator, pest receives a grammar of RTLola specifications as input and generates corresponding Rust code for parsing the specifications into a parse tree. The generated Rust code is utilized by the RTLola frontend to parse specifications which are afterwards transformed into an abstract syntax tree (AST). The AST includes all relevant information from the specification but does not include irrelevant parts like keywords and braces.

In the process of incorporating scheduling annotations, we apply the syntax grammar rules explained in Sect. 3.2 to the pest grammar. First, we extend the RTLola grammar with a rule `ScheduleAnnotation` which corresponds to the nonterminal `sannotation` in Fig. 3.2:

→ Sec. 3.2, p. 29

→ Fig. 3.2, p. 30

```
ScheduleAnnotation = { "schedule " ~ Constraint ~ ("and " ~ Constraint)* }
Constraint = _{ MaxConstraint | MinConstraint | RangeConstraint }
```

```
MaxConstraint = { "max" ~ ":" ~ ConstraintValue }
MinConstraint = { "min" ~ ":" ~ ConstraintValue }
RangeConstraint = { "in" ~ ":" ~ "[" ~ LowerBound? ~ ".." ~ UpperBound? ~ "]" }
LowerBound = { ConstraintValue }
UpperBound = { ConstraintValue }
ConstraintValue = { PriorityConstant | Literal }
PriorityConstant = { "high" | "low" | "medium" }
```

The pest grammar defines a set of rules, the nonterminals. Nonterminals are referenced by their name, while terminals are enclosed in quotation marks. Concatenation of terminals and nonterminals is denoted by the tilde (~) symbol, and a pipe (|) symbol is used to choose from among different alternatives. A question mark (?) makes a rule optional, while the star operator (*) allows zero or more repetitions.

Compared to the syntax rule, the corresponding pest grammar is different in a few places. For one, the pest grammar introduces a lot of intermediate grammar rules. These intermediate rules simplify building the AST from the parse tree produced by the pest parser.

Additionally, the pest grammar allows to mix the inclusion of both durations and frequencies (which are parsed as a `Literal`), as well as priority constraint values:

```
ConstraintValue = { PriorityConstant | Literal }
```

In the syntax for scheduling annotations, however, all constraint values must align with a particular type of constraint value and be used consistently within the specification. Allowing mixing constraint values, and only checking afterward that all constant values align, simplifies the parsing process as a single grammar can parse specifications with all different kinds of constraint values.

Next, we modify the pest grammar to allow the `SchedulingAnnotation` rule to optionally be inserted at the places specified in the syntax:

```
EvalDecl = { "eval" ~ ActivationCondition? ~ (EvalWhen | EvalWith){0,2} ~
    ScheduleAnnotation? }
InputStream = { "input " ~ Ident ~ ParamList? ~ ":" ~ Type ~ ScheduleAnnotation?
    ~(","~ Ident~ ParamList? ~ ":" ~ Type ~ ScheduleAnnotation?)* }
Trigger = { "trigger " ~ ActivationCondition? ~ Expr ~ (StringLiteral ~ IdentList?)?
    ~ ScheduleAnnotation? }
```

→ Fig. 3.1, p. 30

This pest grammar closely follows the railroad diagram Fig. 3.1. One difference is that the pest grammar in the frontend also incorporates RTLola features that are not part of our subset of the RTLola language. These features, like parameters, are not included in the syntax diagram but are present in the pest grammar of the frontend.

4.1.2. Intermediate Representation

The RTLola frontend parses the specification, analyzes it and ultimately produces the MIR. The MIR contains the parsed specification with all the analyzation results and is

used by various RTLola tools and backends. In our approach, we complement the MIR to include the annotated constraints on input streams, output streams and triggers.

Within the MIR, we represent a constraint as the following struct:

```
pub struct Constraint {
    pub constraint: ConstraintVariant,
    pub span: Span,
}
```

A constraint consists of the actual constraint type and a span denoting the constraint's location in the source code. The Span is used to pinpoint error locations during subsequent stages of processing within the translator.

The actual constraint type corresponds to the enum with all possible constraint variants:

```
pub enum ConstraintVariant {
    Min(ConstraintValue),
    Max(ConstraintValue),
    Range(Option<ConstraintValue>, Option<ConstraintValue>),
}
```

Note, that for convenience our design allows a range to not include an upper or lower bound, effectively indicating that the value is not bounded in the direction of the missing bound.

Just as in the parsing stage, our approach allows all different constraint values to be mixed inside the MIR:

```
pub enum ConstraintValue {
    HighPriority,
    MediumPriority,
    LowPriority,
    Priority(u64),
    Duration(Duration),
}
```

This prevents the need to parametrize the frontend based on the specific type of constraint value used. However, when collecting the constraints in the translator, we ensure that all constraint values conform with the required type. A constraint value can be a duration (possibly converted from a frequency), priority, or the high, medium and low priority constants.

In addition to the typical information provided by the RTLola frontend, such as the name or type of a stream, the MIR includes a list of constraints associated with each stream. Exemplary for the input stream, the inclusion of the constraints looks as follows:

```
pub struct InputStream {
    pub name: String,
    pub ty: Type,
    ...
    pub constraints: Vec<Constraint>,
}
```

The resulting MIR is utilized by the translator, which leverages the information to derive a schedule and translate the annotated specification into a regular one.

4.2. Translation

To leverage any existing RTLola backend for scheduled monitoring, our approach involves translating annotated specifications into regular RTLola specifications. The translated RTLola specification can then be monitored seamlessly using a regular RTLola backend, such as the RTLola interpreter or any compiled RTLola monitor.

This section delves into the implementation of the process which translates annotated specifications into regular ones. The translation adheres to the principles outlined in the specification semantics Sect. 3.3, to analyze the annotations and to construct a schedule. This schedule is embedded into the resulting regular RTLola specification as additional output streams. These additional output streams serve as a communication channel between the monitor backend and the scheduler sitting on top which then queries the inputs at the desired times and passes these values to the underlying monitor backend.

→ Sec. 3.3, p. 32

4.2.1. Constraint Values

To facilitate the scheduling of different constraint values (see Sect. 3.1), the translator is generic with the `Scheduleable` trait. This trait encapsulates all functionality specific to a particular kind of constraint value. By structuring the translator in this manner, the rest of the translation process is completely independent of the kind of constraint value used. Additionally, this approach makes it straightforward to add support for additional constraint values.

→ Sec. 3.1, p. 22

Included in the trait are the following important functions:

```
trait Schedulable {
    fn from(v: mir::ConstraintValue) -> Result<Self, ..>;
    fn compare(&self, other: &Self) -> Ordering;
    fn expr(&self) -> Expression;
}
```

In the following, we will explain the three important trait methods in the listing above:

- The `from` method constructs a new constraint value from a constraint value included by the modified frontend in the MIR. As explained in Sect. 4.1, we modify the RTLola frontend as minimally as possible and permit all different kinds of constraint values. Therefore, the MIR returned by the frontend represents all different kinds of constraint values as an enum `mir::ConstraintValue`. The `from` method constructs the concrete constraint value required for the translation progress by extracting the value from the enum. If the enum variant is of the wrong kind than the kind of constraint value needed for the translation, an error is returned instead.

- The `compare` method establishes the comparison between two constraint values, providing the total order as given in the definition of constraint values (see Def. 3.2). → Def. 3.2, p. 22
- As a result of the translation progress, the schedule is included in the resulting RTLola specification as additional streams. To include the respective scheduled constraint values in these streams, a constraint value needs to be expressable as an RTLola expression. This functionality is ensured by the `expr` method.

In the implementation, the `Scheduleable` trait includes some additional methods. These methods are either default implementations provided for convenience such as a greater-than method utilizing `compare`, or serve a debugging purpose, like displaying the current constraint value as a string.

Importantly, the only modification needed to enable the translation of scheduling annotations with a different kind of constraint value is the implementation of this trait. In the following, we will present the implementation of this trait for the two constraint values used in this thesis: durations and priorities.

Durations

A type that implements the `Scheduleable` trait is the `std::time::Duration` type. Obviously, this type represents the duration constraint value introduced in Sect. 3.1.1. → Sec. 3.1.1, p. 23 Consequently, the `from` function is designed to fail if called with a constraint value enum variant different from a duration. If the enum variant does indeed hold a duration, this duration is returned as the constraint value.

To compare two durations, the total order provided in Sect. 3.1.1 is considered: the shorter duration is considered greater, i.e. given a higher priority, when compared with a longer duration. Lastly, durations are expressed in RTLola simply as a floating point number describing the duration in seconds.

Priorities

Another type that implements the `Scheduleable`-trait is the priority. The priority type is a simple wrapper around an unsigned integer, employing the newtype idiom [6]:

```
struct Priority(u64);
```

The use of the wrapper not only permits the future incorporation of other constraint values represented as unsigned integers (potentially with e.g. a different total order) but also enhances clarity by explicitly signifying that the current representation is a priority.

The priority type corresponds to the priority constraint value introduced in Sect. 3.1.2. → Sec. 3.1.2, p. 25 It can be constructed from all variants of the constraint value contained in the MIR, besides durations. For a high, medium or low priority constant (for the syntax of priority constraint values refer to Fig. 3.4), these constants are translated into the corresponding → Fig. 3.4, p. 31

integer value (*low* corresponds to 1, *medium* to 5, and *high* to a priority of 10). Priorities are compared according to the defined total order: higher priority integers are ordered greater than lower priority integers. In RTLola, priorities are simply represented as the unsigned integer itself.

4.2.2. Collecting Constraints

→ Sec. 3.3, p. 32

If we recall the specification semantics from Sect. 3.3, the initial step is to collect all the scheduling annotations into a set. This is also the first step of the translator. The previous section demonstrated the modifications in the RTLola frontend to include the scheduling annotations inside the MIR. The translation process starts with traversing the MIR and extracting all the annotations, along with the event-driven pacing type of the source and the potential filter condition according to Alg. 1. The constraints are represented in the following struct:

→ Alg. 1, p. 46

```
struct Constraint<ConstraintValue: Schedulable> {
    ac: ActivationSet,
    filter: ConstraintFilter,
    constraint: ConstraintVariant<ConstraintValue>,
    span: Span,
}
```

Given that the monitoring approach limits the static filter condition to be a conjunction of inputs, the implementation models the static filter condition as a set of inputs `ActivationSet`. This choice aligns with the specification semantics and facilitates straightforward testing of equality and implications of static filter conditions. The `ConstraintFilter` represents the semantic filter condition of the constraint while `ConstraintVariant` is an enum representing the actual type of constraint (*min*, *max* or *in*) and also holds the corresponding constraint values. The `span` provides information propagated from parsing, specifying the position of the constraint in the specification file. This provides useful information used in the generation of better error messages during the translation.

The entire translation process is parameterized over the constraint value trait `Schedulable`. To construct the constraint value, the `from` method of the `Schedulable` trait is utilized. To enhance error reporting, all constraint values of the wrong kind are collected and displayed as an `RtLolaError`, allowing the RTLola frontend to display a well-formatted representation of all errors along with their respective location in the specification file.

4.2.3. Calculating Schedule

To compute the schedule, the implementation adheres to the algorithm presented as pseudocode in Alg. 4. First, the algorithm groups the constraints based on static pacing types as described in Alg. 2. By representing static filter conditions as sets of inputs, the

→ Alg. 4, p. 50
→ Alg. 2, p. 47

process of testing the implication of static filter conditions simplifies to a superset check between the two set representations.

To construct the schedule, we employ the algorithm outlined in Alg. 3. We categorize the constraints into filtered and total constraints, build the schedule based on filtered constraints, and utilize total constraints to verify the schedule's validity.

→ Alg. 3, p. 48

4.2.4. Resulting Specification

To construct the regular specification that is returned by the translator, the initial step involves removing all scheduling annotations from the annotated specification. Without these annotations, the specification parses as regular RTLola and given that the schedule is already computed, these annotations are no longer necessary. Moreover, we embed the schedule into this resulting specification by introducing additional output streams.

For each schedule with a given static filter, we introduce a separate output stream. These output streams are uniquely named to allow the scheduler to discern which streams contain scheduling information and which streams belong the the original specification. Each added stream shares the same event-based pacing type as the static filter of the schedule it represents. Consequently, the schedule stream is evaluated if the static pacing type is fulfilled, which results in the scheduling of the next deadline for these streams.

For each semantic filter in the schedule, an eval clause is added to the corresponding schedule stream in the same order as in the schedule. The ordering of eval clauses ensures that, when seeking a value to schedule, the first matching constraint value is chosen as defined in the specification semantics (see Sect. 3.3).

To provide a demonstration of these schedule streams, consider the following example:

Example 4.2.1. Assume the following schedule:

$$\text{Schedule}(\{a, b\}) = \langle (a == 0, 5\ \text{s}), (a \neq 0, 10\ \text{s}) \rangle$$

$$\text{Schedule}(\{a\}) = \langle (\top, 8\ \text{s}) \rangle$$

Given that the schedule contains two different static pacing types, the translation progress will add two output streams: `schedule0` and `schedule1`, which are annotated with the corresponding event-driven pacing type and contain the schedule as eval clauses:

```
output schedule0 @(a&&b)
    eval when a = 0 with 5.0
    eval when a ≠ 0 with 10.0

output schedule1 @a
    eval when true with 8.0
```

Consider a scenario where the monitor runs a cycle, receiving 1 as the new value for the input a and 2 as the new value for the input b. Given that both a and b received new values in this cycle, the output stream `schedule0` computes a new value. In the context of scheduling over duration constraint values, this output stream's new value determines when both a and b are required to receive new values again. Given that $a \neq 0$, a and b are expected to receive new values in 10 seconds.

Furthermore, the second output, which exclusively schedules the input a, also calculates a new value. This indicates that a new value for a is required sooner than a new value for both a and b together. The new value for a is already anticipated in 8 seconds. △

4.3. Scheduler

This section is dedicated to the execution of the scheduled monitor. To achieve this, we introduce a scheduler that operates on top of a regular RTLola backend. The scheduler handles querying sensors at appropriate times and passing the new values to the underlying RTLola backend.

The implementation of the scheduler is split into different components. First, we explain the schedule, a component responsible for managing the schedule state and determining which streams are due to update at which times. In Sect. 4.3.2 we introduce a clock, a component able to provide the current time. The interface to the sensors is explained in Sect. 4.3.3. These components are coordinated by the scheduler component, addressed in Sect. 4.3.4.

4.3.1. Schedule

The schedule stands out as the only component within the scheduler that relies on a specific kind of constraint value. It is responsible for storing all the constraint values supplied by the schedule embedded as output streams in the underlying monitor. Moreover, it possesses the capability of providing a timestamp of the next deadline together with a set of inputs due at that time. In the implementation, this functionality is encapsulated within a trait:

```
trait ScheduleRepresentation {
    type ScheduleType;
    fn value_to_schedule_type(value: Value) -> Self::ScheduleType;
    fn schedule(
        &mut self,
        schedule: ScheduledInputsRef,
        with: Self::ScheduleType,
        current_time: Duration,
    );
```

```
    fn next_inputs(&mut self, current_time: Duration) -> (Duration,
        Vec<ScheduledInputsRef>);
}
```

This trait provides the following methods:

- The trait introduces an associated type `ScheduleType`, representing the type of constraint value that is scheduled.
- The trait provides a function to convert the value of an output stream returned by the backend to the associated `ScheduleType`. Given that the value type of the schedule stream is always determined by the kind of constraint value used, this function can never fail.
- The trait includes a method `schedule`, designed to update the schedule for a given set of inputs to the given constraint value. These inputs are derived from the event-based pacing type of the schedule streams added to the specification. The function also takes as a parameter the time of the last monitoring cycle, which was responsible for rescheduling the given inputs.
- Additionally, the trait offers a method to retrieve the next scheduled inputs. The method is responsible for determining which streams are due next and returns all of these streams with the corresponding time of the deadline. This method corresponds to the nextdl^S and pop function of the given constraint value (see Def. 3.2). → Def. 3.2, p. 22

In the following we explore the implementation of this trait for the two kinds of constraint values used in this work: durations and priorities.

Durations

For the duration constraint values, the schedule is straightforward. Given that the schedule streams return durations as floating-point numbers in seconds, the associated `ScheduleType` is also a floating-point number. Consequently, the `value_to_schedule_type` function unwraps a floating-point number from the `Value` returned by the RTLola backend. As it is ensured by the translator for scheduling streams to always have this value type when scheduling over duration constraint values, the function is allowed to panic for every other kind of value.

The schedule stores durations in a priority queue, implemented in Rust's standard library as a `BinaryHeap`. For each scheduled stream, the `schedule` method inserts the corresponding inputs into the binary heap at the designated time. If a specific set of inputs already is scheduled in the queue, the schedule only keeps the entry that is due the earliest.

Subsequently, the `next_inputs` method retrieves all inputs with the lowest deadline from the priority queue, returning them as a set of next inputs along with the associated timestamp of the deadline.

Priorities

Given that priorities are represented as unsigned integers, the `ScheduleType` is accordingly defined as an `u64`. Consequently, the `value_to_schedule` function unwraps an `u64` from the `Value`, panicking for every other value type. As before, this operation is ensured to succeed through the translator.

The scheduling process of priorities diverges from that of durations. To schedule priorities, knowledge of the available bandwidth is needed. This bandwidth is part of the `PrioritySchedule` struct implementing the `ScheduleRepresentation` trait. Additionally, for each stream, minimal and maximal update durations are recorded inside the schedule.

The scheduling process unfolds as follows: during scheduling, the schedule records the current priority assigned to all inputs. When querying for the next scheduled inputs, the process involves iterating through all inputs. Initially, the bandwidth is filled with inputs that are overdue, i.e. those where the maximal update duration has already been reached. Subsequently, if there are remaining spots within the current bandwidth, they are filled with other inputs. This is executed in the order specified in the definition of priority constraint values given in Sect. 3.1.2.

→ Sec. 3.1.2, p. 25

4.3.2. Clock

We allow monitoring in both online and offline scenarios. In an offline scenario we do not want to wait until inputs become due, but just track the elapsed time. This allows us to apply our scheduling approach to the analysis of traces, without actually waiting for inputs in real time. On the other hand in real-world scenarios with actual sensors, waiting in real-time is crucial.

To accommodate both scenarios, we employ a clock interface:

```
pub(crate) trait ClockRepresentation {
    fn current_time(&self) -> Duration;
    fn wait_until(&mut self, until: Duration);
}
```

A clock provides methods to retrieve the current time and to wait until a specified time. Depending on the nature of the monitoring scenario, the clock updates in real-time or merely advances a simulated internal time instantly.

4.3.3. Sensors

Unlike regular RTLola, where the values for input streams are provided automatically from the outside, the scheduled approach requires the possibility of querying new values for input streams. However, the origin of these input values varies depending on the application. For analyzing traces, the appropriate value is simply retrieved from

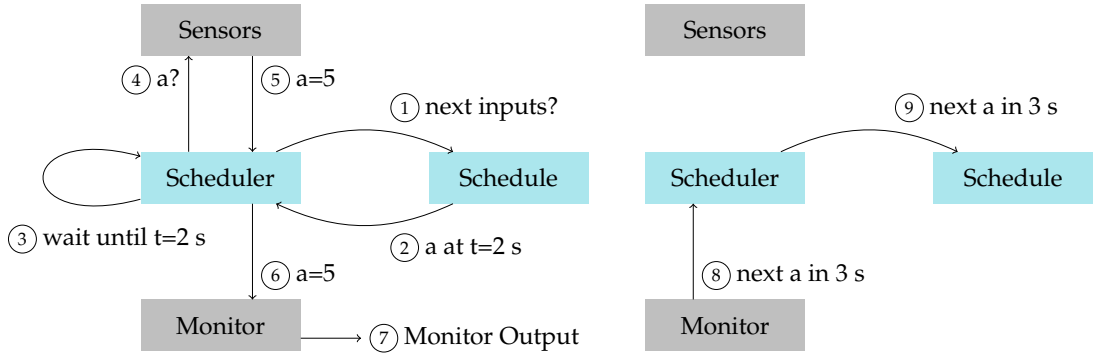


Figure 4.3.: Depiction of one cycle of the scheduler. The *scheduler* is the central component responsible for querying values from the *sensors*, storing scheduling information in the *schedule* and running the underlying *monitor* backend. Numbers represent the sequential order of steps during the cycle.

a previously recorded trace. On the other hand, for applications involving real sensors, the scheduler has to communicate with the sensor to obtain the current value.

To accommodate these diverse applications, we introduce the `InputSource` trait. This trait defines the method `query`, which, given an input reference, retrieves the value of that input at the specified time:

```
trait InputSource {
    fn query(&self, input: InputReference) -> Option<Value>;
}
```

This trait provides flexibility by allowing implementations specific to the needs of the current application. In Chapter 5, we present a `CsvSource` for retrieving values from a trace in CSV format and a source retrieving values from simulated sensors of a drone.

→ Chapter 5, p. 67

4.3.4. Scheduler

The scheduler is the central component controlling the other components introduced above. It is responsible for querying data from sensors, updating the internal schedule, and forwarding sensor readings to the underlying monitor backend at the appropriate times.

Similar to regular RTLola, the scheduler operates in cycles. Fig. 4.3 illustrates a single cycle of the scheduler. The *Sensors*, the top box in the graphic, are accessible to the scheduler through the `InputSource` trait introduced in Sect. 4.3.3. The right component labeled *Schedule* is available to the scheduler through the `Schedulable` trait introduced in Sect. 4.3.1. The bottom component labeled *Monitor* is the underlying monitor backend, which, in this implementation, is the RTLola interpreter [7]. However, it is easily adaptable to any other RTLola backend.

The numbers on the edges denote the sequential order of steps occurring during one cycle. Specifically, the individual steps depicted in the graphic are detailed as follows:

(1) At the beginning of the scheduler cycle, the scheduler unit retrieves inputs due at the next deadline using the `next_inputs` method of the *ScheduleRepresentation*.

(2) The schedule returns a list of inputs along with a timestamp indicating when the inputs are due. For example, in the graphic, input a is due at the absolute time of 2 seconds.

(3) The scheduler waits until the next deadline specified by the schedule, utilizing the `wait_until` method provided by the *Clock* (see Sect. 4.3.2).

(4) When the time of the deadline is advanced, the scheduler queries the sensors for the current values of the desired input streams with the `query` method of the *Sensors*.

(5) The sensors provide the new values for these inputs. The graphic assumed the current value of a at the absolute time of 2 seconds to be 5.

(6) This new value is passed to the underlying monitor backend.

(7) Subsequently, one regular cycle of the monitor is executed by the underlying monitor backend. The regular output of the monitor is produced and forwarded to the user.

(8) Simultaneously, the new values of the schedule streams are communicated back to the scheduler component. The new values specify the new constraint values scheduled for those inputs. In this example, assume that the schedule embedded as schedule streams in the underlying monitor decided that input a is due to receive a new value in 3 seconds.

(9) The scheduler stores this information in the schedule. This is done by utilizing the `schedule` method of the *ScheduleRepresentation* trait.

(10) A new cycle begins, starting again with (1).

In conclusion, the scheduler emerges as a central component, coordinating the flow of information between sensors, the schedule and the underlying monitor backend.

4.4. Timing Considerations

In this section, we elucidate the challenges inherent in the practical implementation of this monitoring approach. The majority of these challenges are linked to timing considerations in online-monitoring applications. First, we discuss the issue of latency encountered when retrieving new values from sensors. Subsequently, we explore a method to address missing input values and input values arriving too late.

4.4.1. Query Time Delay

In praxis, querying new values from a sensor requires some amount of time. The theory, however, assumes instantaneous receipt of all inputs precisely when they are needed. Given that this is not possible in praxis, the scheduler must account for the temporal delay between the query of a new value and the arrival of those values.

To address this challenge, the scheduler must query for new values before they are due, allowing sufficient time for them to arrive on time. However, the querying time varies across systems and, often, among individual sensors. Consequently, the scheduler measures the time elapsed from the initiation of a query until the arrival of the corresponding data for each input. Based on this timing information, the scheduler calculates a mean querying time and considers the median querying time when querying for new values.

However, querying times may exhibit substantial variation between consecutive queries, posing a challenge to achieving precise timing. While it may be challenging to completely eliminate this variation, the mean or median querying time might still be acceptable when allowing the values to arrive within some tolerance compared to the scheduled time.

4.4.2. Missing Inputs Values

The potential for sensors malfunctioning introduces additional challenges, as inputs may either arrive too late or not at all. Detecting and responding to such deviations from expected behavior is critical for the monitoring system. The monitor must be able to detect such cases and apply appropriate actions to fix the issue. Such actions may be simply requesting the value from the sensor again, alerting the user to fix sensor issues, or even the shutdown of the entire system if deemed necessary.

In order to allow the system to react to missing or delayed input values, we can introduce additional triggers to the specification, designed to validate the timely arrival of scheduled inputs:

```
input time : Float64
input scheduled_for : Float64
trigger abs(time - scheduled_for) > TOLERANCE "inputs did not arrive on time"
```

Two new input streams have been incorporated into the specification to support this functionality. The first input stream, `time`, contains the current time of the monitor and is provided by the RTLola backend itself. The second input stream, `scheduledK_for`, contains the deadline for which the current set of input values is scheduled to receive new values. Leveraging these two information, we define a trigger checking the difference between both timestamps to not be larger than a predefined tolerance.

With this approach, the user is granted the flexibility to determine the appropriate course of action when this trigger activates, whether it involves logging a warning or opting for a complete shutdown of the system.

Chapter 5

Evaluation

In this chapter, we conduct an evaluation of the scheduled monitoring approach introduced in Chapter 3. Sect. 5.1 evaluates the scheduled monitor using a simple, generated trace, allowing the results to be easily visualized. Afterward, in Sect. 5.2, we employ a simulator to apply the scheduled monitor in an online setting monitoring a drone.

5.1. Generated Trace

As the initial step of our evaluation, we execute the scheduled monitor using a generated trace. The employed specification is intentionally simplistic, involving only a single input. This approach facilitates a clear visualization of the scheduler's queries, enabling a visual comparison between the scheduled and unscheduled monitor executions for this specification.

This section first introduces the specification and afterwards the generated trace. Finally, we present visual representations of the monitoring runs and compare both of them.

5.1.1. Specification

For the first part of the evaluation, we opt for a specification featuring just one input. Hence, we have chosen to simplify the example of a geofence to a one-dimensional geofence, constraining a value within specified upper and lower bounds:

```
input a : Float64

constant LOWER_BOUND : Float64 := 0.0
constant UPPER_BOUND : Float64 := 5.0

output outside_bounds := a ⩽ LOWER_BOUND ∨ a ⩾ UPPER_BOUND
```

```
output move_outside :=
    outside_bounds ∧ ¬outside_bounds.offset(by:-1).defaults(to:false)
trigger move_outside "the value a moved outside the geofence"
```

The specification consists of a single input stream a and constrains its value to fall within the range of 0 as the lower bound and 5 as the upper bound. Whenever the value of a surpasses these boundaries and was not outside before, the specification issues a trigger.

The subsequent stage involves the introduction of scheduling annotations, aiming to prompt queries for the next value of a more frequently when the current value is near the specified boundaries, but less frequently when it is far away. To achieve this, we introduce a supplementary stream named `distance_to_bounds`, which computes the proximity to the nearest bound:

```
output distance_to_upper_bound := UPPER_BOUND - a
output distance_to_lower_bound := a - LOWER_BOUND
output distance_to_bound := min(distance_to_upper_bound, distance_to_lower_bound)
```

We incorporate an additional stream that is assigned a different scheduling constraint depending on the distance to the bounds. This stream is then utilized in the trigger. Given that we are dealing with only one input, it is not meaningful to add priority constraints. Consequently, the annotations schedule the duration for the next value of a:

```
output outside_bounds_trigger
    eval @a when distance_to_bound ⩽ 0.6 with move_outside schedule min: 0.8s
    eval @a when distance_to_bound ⩽ 0.8 with move_outside schedule min: 1.5s
    eval @a when distance_to_bound ⩽ 1.2 with move_outside schedule min: 3.0s
    eval @a with move_outside schedule min: 4.0s

trigger outside_bounds_trigger "the value a moved outside the geofence"
```

The trigger's expression remains constant, checking whether the new value results in moving outside the bounds. However, the scheduled time for the next value varies based on the last received value:

- If the distance to the nearest bound is less than or equal to 0.6, a new value for a is required in at most 0.8 seconds.
- If the distance to the nearest bound is less than or equal to 0.8 (and the previous statement did not already match), a new value for a is required in at most 1.5 seconds.
- If the distance to the nearest bound is greater than 0.8 and less than or equal to 1.2, a new value for a is required in 3.0 seconds.
- If the distance to the nearest bound is greater than 1.2, a new value for a is only required in 4.0 seconds.

→ Appendix A.1, p. 87

The entire annotated specification can be found in Appendix A.1.

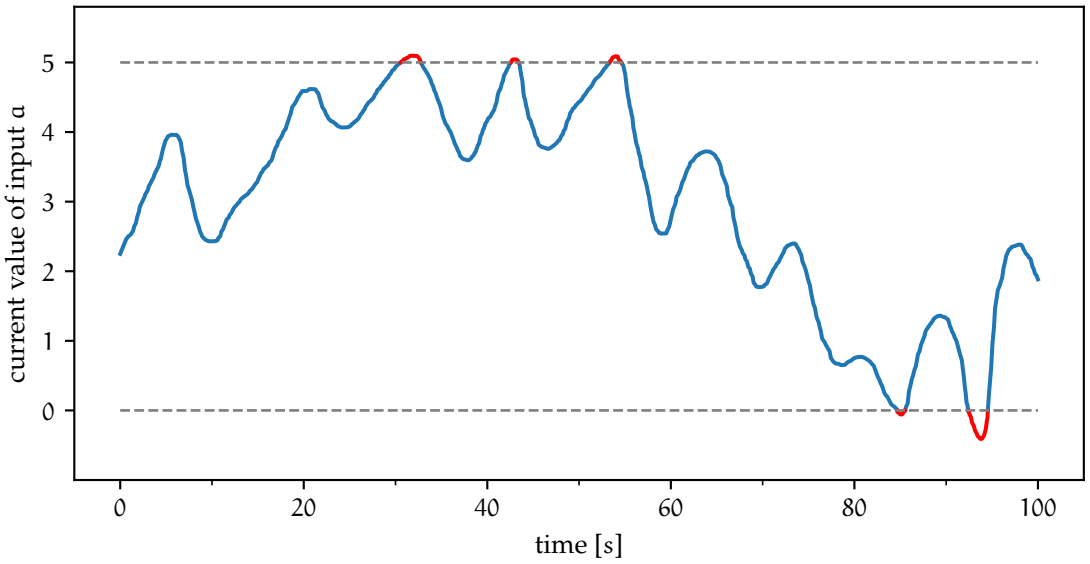


Figure 5.1.: The generated trace represented as a graph. The x-axis corresponds to time in seconds, while the y-axis reflects the current value of the input a. The gray horizontal lines indicate the upper and lower bounds. Instances where the trace violates the specification, i.e. extends beyond the specified bounds, are highlighted in red.

5.1.2. Trace

We generated the trace by capturing the y-coordinate of the mouse position over 100 seconds, with intentional up and down movements of the mouse inside a region on the desktop representing the upper and lower bound of the geofence. This straightforward method enables the creation of scenarios where the geofence is briefly crossed, almost crossed, and, overall, allows producing a trace that is informative for the evaluation and comparison between regular and scheduled RTLola monitoring. The graphical representation of this trace is depicted in Fig. 5.1.

5.1.3. Input Source

In this scenario, it is not necessary to interact with real sensors. Instead, the scheduler retrieves the inputs from a pre-recorded trace of values. For this use case, we introduce the `CsvSource`, which implements the `InputSource` trait introduced in Sect. 4.3.3.

→ Sec. 4.3.3, p. 62

Internally, the `CsvSource` operates by opening a trace file in CSV format and seeking the appropriate location based on the current time of the scheduler. Given that the rows in trace files are sorted ascending by time, the `CSVSource` marks its current position in the trace and advances it as time progresses in the scheduler. In situations where an

	Fixed Frequency	Scheduled
Number of Inputs	51	51
Activated Triggers	3	5
Useful Inputs (%)	5.88%	9.80%

Table 5.3.: Comparison between monitoring with a fixed frequency and with the scheduled approach on the monitoring run depicted in Fig. 5.2.

exact entry at the specified time is not available, the source opts for the most recent recorded value preceding that specific time.

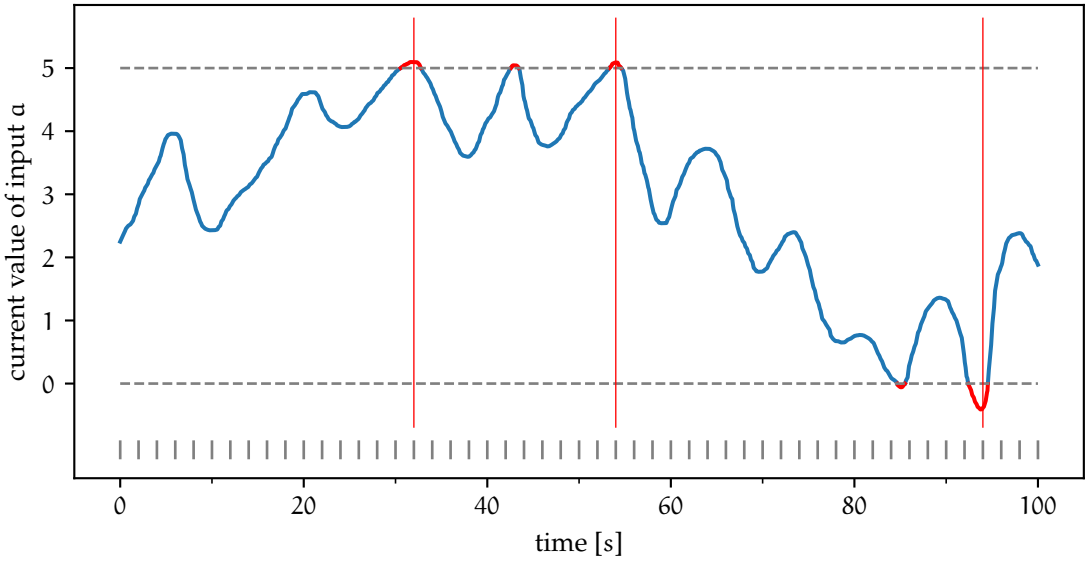
5.1.4. Results

We execute the specification without any scheduling annotations, providing the new value of the input a at a fixed frequency of 0.5 Hz, i.e. every two seconds. The results of this run are illustrated in Fig. 5.2a. In the graphic, a red vertical line signifies trigger activations and small gray lines at the bottom indicate instances when the monitor receives a new input value. Notably, certain segments of the trace violate the geofence for such a brief duration that the frequency at which new values arrive in the monitor is too sparse for the monitor to detect the violation.

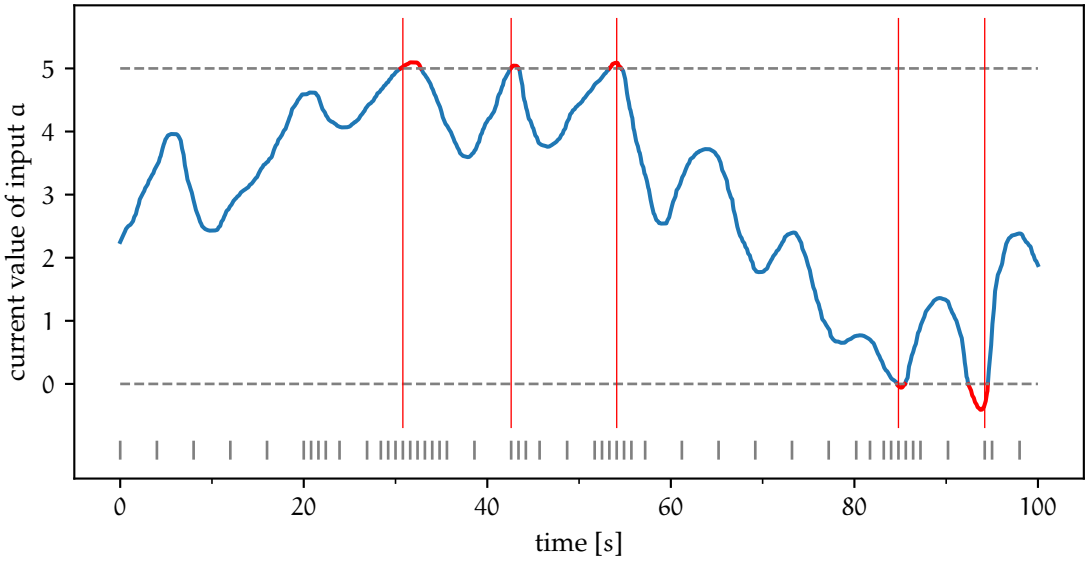
For scheduled monitoring, we translate the annotated specification shown above into RTLola without annotations using the translator. The resulting specification can be found in Appendix A.2. Subsequently, we execute the resulting specification with the scheduler in offline mode, utilizing the trace introduced earlier. The results of the scheduled monitoring run are depicted in Fig. 5.2b. Given the gray marks at the bottom of the plot, it is evident that the sensor is queried much more frequently when the value is near the bounds compared to when it is far away. Upon crossing the bounds, the value immediately preceding the crossing was already in close proximity to the bounds, thereby leading to a frequency large enough to successfully detect all violations of the geofence.

→ Appendix A.2, p. 88

Tbl. 5.3 depicts information regarding both monitoring runs. Although the number of input values is identical, the scheduled monitor successfully detected all 5 violations, whereas the regular monitor with a fixed frequency only managed to detect 3 violations. When calculating the ratio of new input values that resulted in a violation of the specification, we observe that the scheduled monitor makes wiser choices on how to use the equal number of input values. Specifically, the regular monitor identified a violation in 5.88% of the queries, whereas the scheduled monitor detected violations in 9.8% of all queries.



(a) Regular RTLola monitor receiving inputs with a fixed frequency of 0.5 Hz.



(b) Scheduled RTLola monitor receiving inputs according to scheduling annotations.

Figure 5.2.: Comparison between monitoring with a fixed frequency and with the scheduled approach. The red vertical lines indicate triggers, while the gray marks at the bottom depict the times when the input a receives a new value.

The scheduled approach may seem less meaningful with just one input because utilizing a regular monitor and increasing the querying frequency would result in finding all violations as well. However, the regular monitor would require significantly more values of α to identify these violations. The advantages of the scheduled approach become apparent in setups involving multiple inputs and limited bandwidth. In those scenarios, it is not possible for all inputs to receive new values at such a high frequency. With the scheduled approach, the scheduler would opt to utilize the available bandwidth for data from other sensors when the geofence does not currently require frequent new inputs because it is very unlikely to produce a violation soon.

Moreover, the specification reveals a challenge with scheduling duration constraint values. Finding suitable constraints and when-conditions which together result in a number of queries comparable to the regular monitor with a fixed frequency required some trial and error and led to an unintuitive specification. The issue would persist when scheduling multiple inputs and aiming to optimize bandwidth usage efficiently, as it is not straightforward to formulate scheduling annotations over duration constraint values that make the most efficient use of available bandwidth. One solution to this challenge is to schedule based on priorities instead of durations. This adjustment would allow the scheduler to determine the timing of inputs, making the specification-writing experience more intuitive. We will explore scheduling using priority constraint values in the next chapter.

5.2. Simulator

In the second phase of the evaluation, we want to assess the scheduled monitoring approach in an online setting. For this, we deploy our approach in the context of monitoring a drone by leveraging the AirSim simulator. We start the following sections with an introduction to the AirSim simulator and discuss the evaluation setup afterward in Sect. 5.2.2. In Sect. 5.2.3 we discuss the specification employed for the evaluation and finally present the obtained results of the evaluation in Sect. 5.2.4.

5.2.1. AirSim

For this evaluation, we use AirSim [8], a simulator for drones and cars developed by Microsoft. It leverages the Unreal Engine and exposes access to the simulator through an API, allowing access to current data from the simulation and enabling control over the vehicles. The vehicles can be equipped with a diverse array of different sensors [9], such as a camera, a barometer, an inertial measurement unit, a GPS Sensor, a magnetometer, a distance sensor and a lidar sensor. Beyond API interaction, AirSim allows controlling the vehicles using the keyboard or an Xbox controller (see Fig. 5.4a), while the simulated environment is visually represented through an Unreal Engine window (see Fig. 5.4b).

(a) Controlling a drone in AirSim using an Xbox controller.

(b) Screenshot from the AirSim simulator.

Figure 5.4.: Images from running the AirSim simulator.

5.2.2. Setup

For the evaluation, our objective is to compare the performance of a scheduled monitor with monitors querying all inputs with a fixed frequency. To allow a meaningful comparison of the results, all monitor instances are run on the same flight of a drone. For this, all monitors are executed concurrently on the same AirSim simulation.

Fig. 5.5 illustrates the setup employed for the evaluation using AirSim. An annotated RTLola specification is monitored using different monitor instances. One of these monitors adheres to our scheduled approach, while the remaining monitors query all inputs with a fixed frequency. Each of these additional monitors queries the inputs with a different frequency, allowing us to compare the advantages and disadvantages of higher and lower frequencies. During the monitor progress, logs of the monitoring runs are recorded, allowing us to later analyze the results. An Xbox controller is employed to manipulate the drone's flight, while all monitors concurrently observe its sensors in parallel.

Access to the drone's sensors is done through the API provided by the simulator. The input source of the scheduler is designed to hold a TCP connection with the simulator, allowing the input source to query the appropriate sensor using the API once requested by the scheduler.

5.2.3. Specification

For the evaluation with AirSim, our goal is to model a scenario involving a drone conducting experiments with a set of sensors, exemplified here by the barometer. Additionally, the drone has a geofence that dictates the permissible altitude and distance from the starting point. In this context, the barometer should receive as much data as

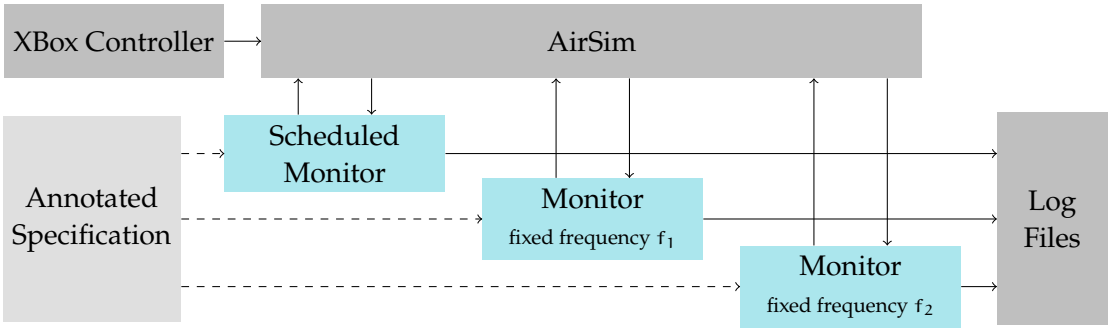


Figure 5.5.: For the evaluation in an online setting, multiple monitors are run in parallel. The annotated specification is monitored with the scheduled monitor, and simultaneously with a set of monitors querying all inputs with a fixed frequency. All monitors query the input values from the simulator through the AirSim Sensor API, while the simulator is controlled using an Xbox controller. Meanwhile, each monitor stores log files which are analyzed afterward.

possible, ensuring that the experiments receive substantial data. However, a violation of the geofence is also really important and should be detected as fast as possible.

To address this, the specification assigns the barometer inputs a permanent medium priority, while the priority of the inputs concerning the geofence is dynamically chosen depending on the distance to the geofence. If the drone is far away from the border of the geofence, the priority of the corresponding input is low, and the monitor assigns a high priority to the corresponding input, if the drone is near a violation. This way, the monitor favors the barometer experiments when the drone is far from violations but prioritizes detecting a geofence violation as fast as possible when the drone approaches potential violations.

The specification consists of four inputs: `gps_lat_long` and `gps_altitude`, dedicated to detecting violations related to the geofence, and two inputs designated for the experiments: `barometer_pressure` and `barometer_altitude`. Leveraging the coordinates in `gps_lat_long`, the specification calculates the horizontal distance to the drone's starting location, and defines a trigger with the following priorities:

```
output geofence := distance_to_start ⩾ 8.0

output scheduled_geofence
    eval @gps_lat_long when distance_to_start ⩽ 4.0
                       with geofence schedule min: low
    eval @gps_lat_long when distance_to_start ⩽ 6.0
                       with geofence schedule min: medium
    eval @gps_lat_long with geofence schedule min: high

trigger scheduled_geofence "outside geofence"
```

Additionally, the specification defines a trigger concerning the allowed altitude above the (landed) starting location:

```
output altitude_bound := altitude_above_ground ⩾ 10.0

output scheduled_altitude_bound
    eval @gps_altitude when altitude_above_ground ⩽ 5.0
                      with altitude_bound schedule min: low
    eval @gps_altitude when altitude_above_ground ⩽ 7.5
                      with altitude_bound schedule min: medium
    eval @gps_altitude with altitude_bound schedule min: high

trigger scheduled_altitude_bound "altitude too high"
```

The complete annotated RTLola specification can be found in Appendix A.3. → Appendix A.3, p. 89

5.2.4. Results

For the evaluation, we executed a scheduled monitor on the specification given above with a *bandwidth of 2 values every 0.5 seconds*. Inputs are considered *overdue after not receiving a new value for 3 seconds*. Additionally, we ran 4 regular monitoring instances for the same specification, each querying all inputs at a fixed frequency of 2 Hz, 1 Hz, 1/2 Hz and 1/3 Hz respectively.

In the following section, we first highlight the results of a single monitoring flight. Afterward, we analyze the monitoring across a number of different flights.

Single Flight

The results of the monitoring of a single flight with the simulator are depicted in Fig. 5.6. This figure comprises two plots illustrating the current values of the outputs `distance_to_start` and `altitude_above_ground` over time. The specification defines an upper bound for both of these streams, beyond which it is considered a violation of the specification. These upper bounds are represented in the graph as horizontal dashed lines.

Below the plot, we display the moments when input streams receive a new value. Each horizontal bar indicates the timestamps of new values for an input stream. The first four bars correspond to the four inputs to the scheduled monitor, with the background color indicating their current priority assigned by the scheduler - green for a low priority, yellow for a medium priority and red for a high priority. The bottom four bars signify the times when the four fixed-frequency monitors received new inputs, where each marking corresponds to the moment when all inputs of that monitor were updated.

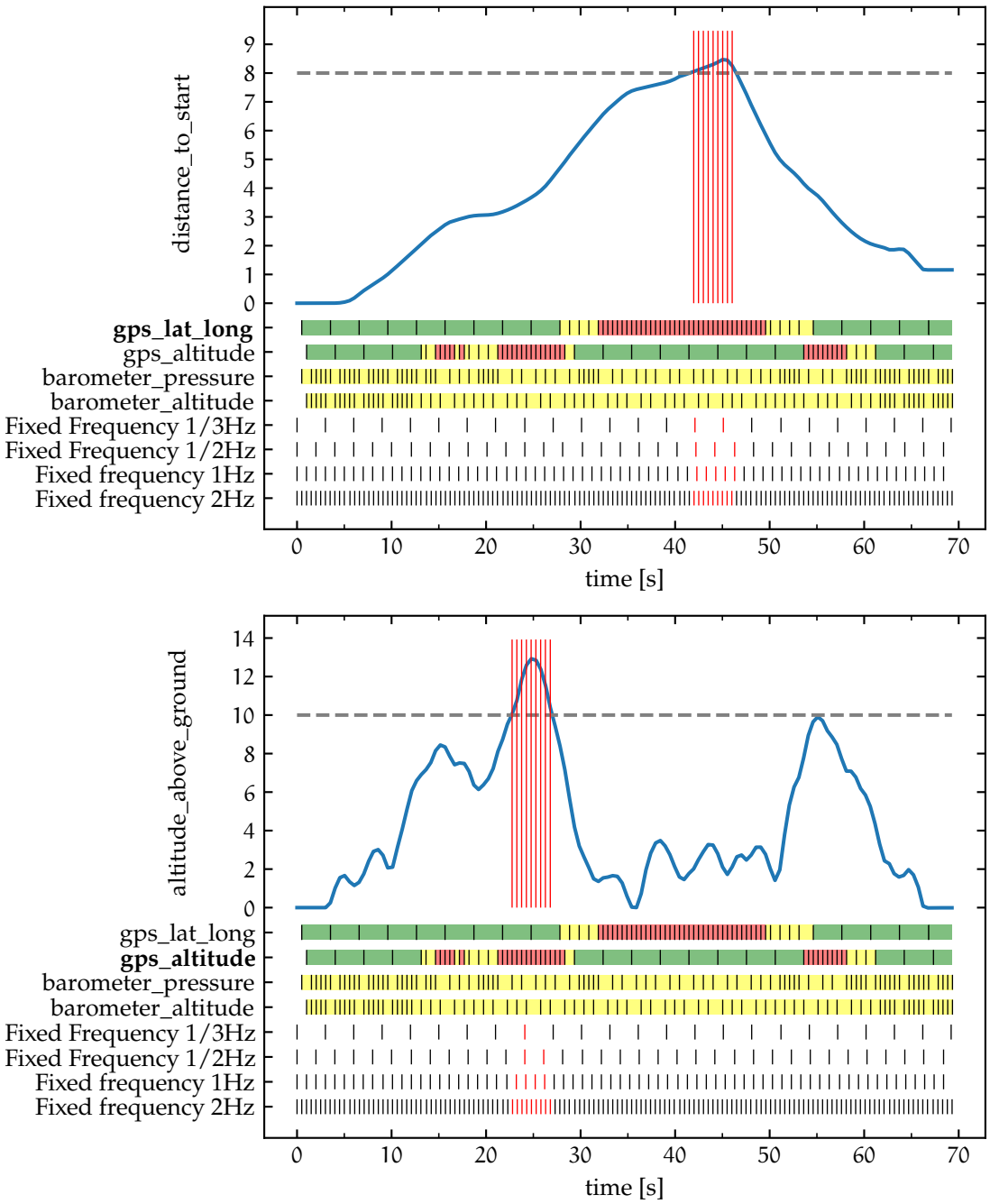


Figure 5.6.: The value of the output streams `distance_to_start` and `altitude_above_ground` during the same flight with the simulator. Below the graph, the instants of new values and the priorities of the scheduled approach are indicated. Vertical red lines indicate a trigger violation found by the scheduled monitor for the corresponding output stream.

Vertical red lines on top of the plotted graph denote each instance when the scheduled monitor issued a trigger for the corresponding output stream. For the fixed-frequency monitors, the markings depicting the timestamps of new inputs are colored in red, if the corresponding monitor issued a trigger at that time for the corresponding output stream.

The graphical representation reveals that consistent with the chosen bandwidth of 2 values every 0.5 seconds, always two inputs of the scheduled monitor receive a new value at the same time. However, the selection of inputs depends on the current priority of the inputs. Initially, with low altitude and distance values, these streams are assigned a low priority. Consequently, barometer inputs, possessing higher priority, are queried more frequently. Nonetheless, inputs with low priority are periodically queried due to their overdue status after 3 seconds. As the scheduled output stream approaches a violation, the diagram depicts an increase in the assigned priority for the corresponding input stream. Because of the higher priority, the available bandwidth is used differently: Now, more of the bandwidth is used on the input that might lead to a violation, than on the inputs with lower priority.

For the marker of the fixed frequency monitors, we can see that in many cases the monitors with lower frequency identify violations later than monitors with a higher frequency. However, it is also noteworthy that monitors with a high fixed frequency query substantially more data than the scheduled monitor. Despite this, the scheduled monitor consistently detects violations comparably early, even when compared to the monitor with the fastest frequency.

Fig. 5.7 provides additional details on this observation. The diagram compares the values obtained by two monitors, each monitoring the output stream `altitude_above_ground` during the flight depicted in Fig. 5.6. The left side of the diagram illustrates the values retrieved by the monitor operating at a fixed frequency of 1/3 Hz, while the right side displays the values obtained through the scheduled approach. In the background, a gray line traces the values recorded by the monitor with the highest fixed frequency of 2 Hz. A vertical red line indicates the first time a trigger violation was found by the corresponding monitors.

As depicted in the diagram, it is evident that the step size of the left monitor is too large, hindering its ability to promptly detect violations. In contrast, the scheduled approach exhibits a large step size when far from the specified bounds; nevertheless, it dynamically adjusts this step to be sufficiently small to find the violation very quickly. Notably, the monitor with the lowest frequency identified this violation 1.3 seconds after the scheduled monitor found the violation.

Statistical Insights Across Multiple Flights

Next, we highlight some statistical insights from the evaluation. To achieve this, we extend our analysis beyond the flight presented in Fig. 5.6 to include nine additional flights, which are illustrated in Appendix A.4.

→ Appendix A.4, p. 90

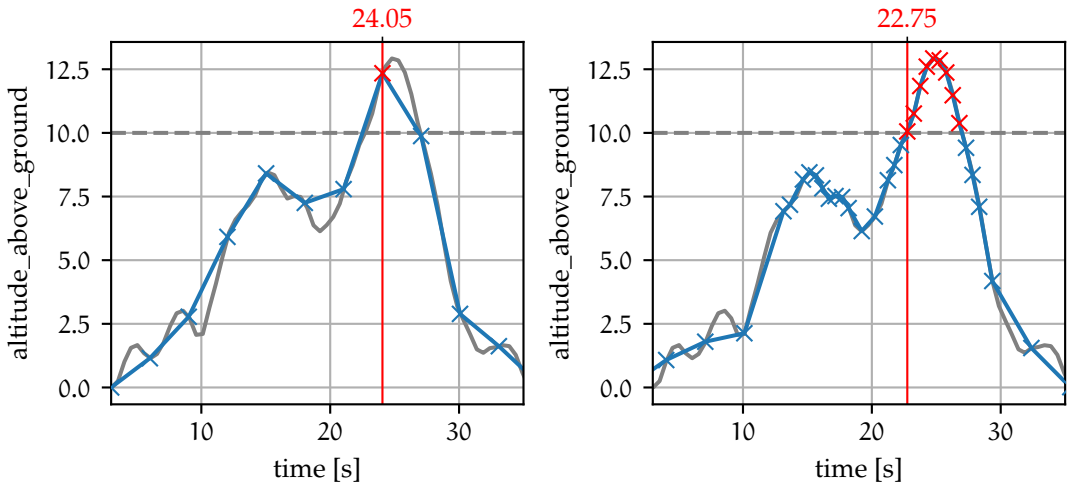


Figure 5.7.: Comparison of values of the `altitude_above_ground` stream queried by the monitor with a fixed frequency of 1/3 Hz (left) and the scheduled monitor (right). In the background in gray, the values obtained by the monitor with a fixed frequency of 2 Hz. A vertical red line denotes the moment when the monitor identified the trigger violation for the first time.

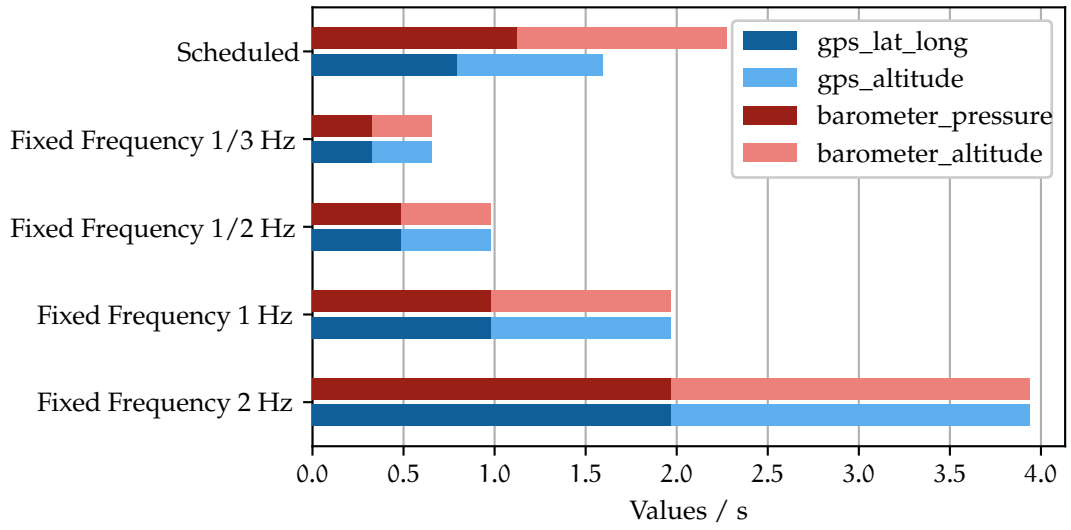


Figure 5.8.: Comparison of the average bandwidth used for the four input streams by the different monitors over 10 drone flights with the simulator. The inputs are grouped into inputs concerned with the experiment (red) and inputs concerned with the geofence (blue).

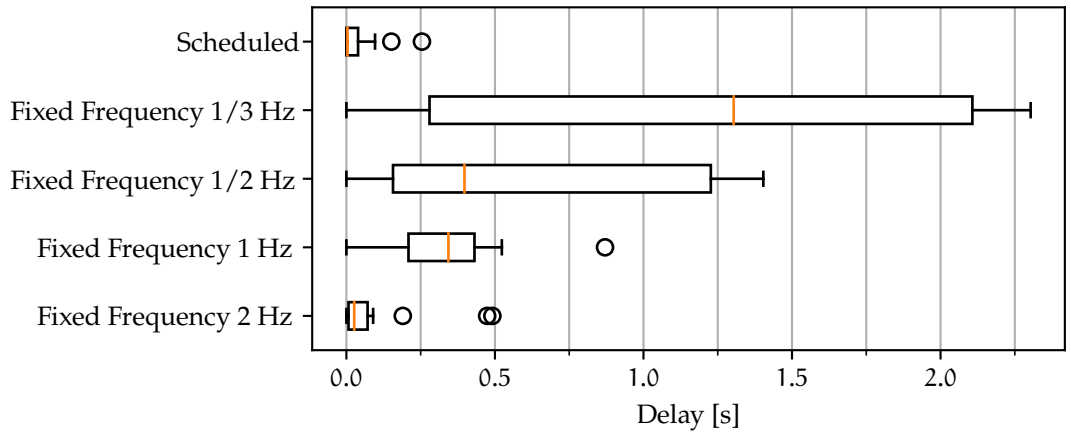


Figure 5.9.: Boxplot illustrating the delay, measured after the detection by the first monitor, until the corresponding monitor identifies a violation of the specification for the first time across the 10 drone flights.

First, we plot the average bandwidth utilized by each monitor for the four input streams, as depicted in Fig. 5.8. In this plot, inputs are grouped into inputs associated with the experiment (depicted in red) and those related to the geofence (depicted in blue). Notably, monitors with fixed frequencies query all inputs at the same rate. In contrast, the scheduled approach exhibits a distinction by querying more inputs related to the experiment than inputs related to the geofence. This is because the scheduler intended to maximize the data collection for the experiments, while still ensuring rapid detection of all geofence violations. However, for the barometer inputs that share the same constant priority, the scheduled approach is fair by allocating an equal amount of bandwidth to both inputs.

The graphic presented in Fig. 5.9 illustrates a boxplot portraying the delay after which a monitor identifies a trigger violation for the first time, after the initial detection of that particular violation by the first monitor. Instances, where a specific monitor fails to identify a particular violation, are excluded, as well as streams where no violations occurred throughout the entire flight.

To demonstrate the information conveyed by the boxplot, consider the following example:

Example 5.2.1. Consider the trigger violation depicted in Fig. 5.7. Tbl. 5.10 provides the times at which this violation was first detected by any monitor. In the example, the violation was identified first by the scheduled monitor, hence assigned a delay of 0 s. Each subsequent monitor is assigned the duration since the first violation, with the monitor at a fixed frequency of 1/3 Hz for example having a delay of $24.05\,\text{s} - 22.75\,\text{s} = 1.3\,\text{s}$. △

Monitor	First Violation	Delay
Scheduled	**22.75 s**	0.0 s
Fixed Frequency 1/3 Hz	24.05 s	1.3 s
Fixed Frequency 1/2 Hz	24.12 s	1.37 s
Fixed Frequency 1 Hz	23.16 s	0.41 s
Fixed Frequency 2 Hz	22.76 s	0.01 s

Table 5.10.: Comparison of the timestamps for the detection of the trigger violation depicted in Fig. 5.7 by each monitor.

The boxplot reveals that the scheduled monitor performs comparably to the monitor with a fixed frequency of 2 Hz in detecting violations. Notably, when considering bandwidth usage as depicted in Fig. 5.8, the scheduled monitor achieves this with significantly less bandwidth consumption. In contrast, the monitor with a fixed frequency of 1 Hz, despite utilizing a comparable amount of bandwidth as the scheduled approach, exhibits a considerably larger delay in identifying trigger violations.

Chapter 6

Related Work

Earlier approaches for runtime monitoring are often based on temporal logics. The Temporal Rover [10] is a commercial tool to monitor a lot of different languages with the specification inlined into the source code. The tool modifies the source code in a way that it behaves like before, only that the specification is checked during the runtime of the program. PathExplorer [11], a monitoring tool developed by NASA, is another famous example. Here the temporal logic specification is given in a separate file.

In our approach, the specification is not given in a temporal logic, but instead we are using a stream-based specification language. The earliest application of stream-based languages were synchronous programming languages such as Lustre [12], Esterel [13] and Signal [14]. One of the first stream-based monitoring language was Lola [2]. Lola later evolved into RTLola [1], which adds real-time features such as sliding windows to Lola. Another stream-based specification languages is TeSSLa [15].

The language we are using in our approach, RTLola [1], was already used in many different applications. It was successfully used to monitor unmanned aircraft during flight and also to analyze their log files afterward [16]. Besides that, RTLola was used to analyze network traffic detecting fingerprinting attacks [17], monitor smart contracts by compiling a monitor from an RTLola specification to Solidity [18] and monitor web-based auction systems [19]. Biewer et al. [20] present an Android app that uses RTLola to monitor the exhaust emissions of a car.

All the mentioned monitoring approaches have in common that they do not take the available bandwidth into account. When there is less bandwidth available than needed to fully monitor the system with the complete specification, some kind of scheduling is required. A different area of research where scheduling is really important, lies in assigning machine code to cores inside a processor to make it as efficient as possible. A lot of research takes place in this area, with some approaches also taking the bandwidth into account e.g. [21, 22, 23, 24].

Another example where the available bandwidth is important are wireless sensor networks. In a wireless sensor network, a lot of different nodes are wirelessly connected as a network and can only talk to neighboring nodes. When monitoring different sensors that are part of such a network, it is important to take the bandwidth into account. But unlike our setup, in such a network it is also possible to split the monitoring between different nodes and to decide which information takes which path through the network. Bhuiyan et al. [25] analyze the frequency content of the incoming data and query the sensor in a frequency according to the current data. Additionally, uninteresting sensor readings are not transmitted through the network and are therefore reducing the used bandwidth.

To allow scheduled monitoring, the monitor can not expect to receive information about every change in the system, as a lot of monitoring approaches do. Instead, the monitor has to periodically sample the current state of the system. Bonakdarpour et al. [26] suggest a time-driven approach for runtime monitoring. To reduce the overhead of monitoring, the system is sampled in a previously calculated frequency. In contrast to our approach, here the sampling frequency is the same during the whole runtime of the program and is calculated in a way that ensures that no violation of the specification is missed.

Using the same sampling frequency during the whole runtime of the program forces the monitoring to choose the highest sampling frequency needed for any part of the program, even though it is only needed in very small parts and the vast majority of the execution would require a much smaller frequency. Navabpour et al. [27] build upon their previous work to make it path-aware. They do not use the same sampling frequency for the whole monitoring, but allow the frequency period to change according to the current state of the system. This reduces unnecessary samples in periods of the program execution that would allow for a much smaller frequency. They also use symbolic execution to predict the program's behavior and adjust the sampling period accordingly. In contrast to our approach, even when the sampling period can change, it always samples the whole system and can therefore not prioritize inputs to the monitor and sample some inputs with a higher sampling rate than others.

Stoller et al. [28] go a similar approach to reduce the monitoring overhead. They sample the program state periodically and use Hidden Markov Models to fill in and estimate the current state in the gaps. The approach of Huang et al. [29] adapts the amount of monitoring to a user-supplied target overhead. This approach is also close to ours since it adapts the monitoring to a target amount of overhead while we adopt the monitoring to a maximal bandwidth. In contrast to our approach, this approach does not sample the state periodically, but temporarily disables the monitoring of selected events that would overwise exceed the target overhead.

In our approach, we are building a scheduler on top of an RTLola monitor. For the underlying monitor, multiple RTLola backends could be used. The RTLola interpreter [7] directly executes the specification. It is also possible to compile the RTLola

specification into VHDL and run the monitor on an FPGA [30]. VHDL has the advantage that it describes hardware circuits which results in a very fast and energy-efficient implementation. There exists a verifying compiler for Lola [31], which generated highly parallel Rust code with verification annotations, so that the correctness of the monitor can be verified automatically. Additionally, there exists a universal RTLola compiler, which makes it easy to translate an RTLola specification into different target languages. The generated code can also contain verification annotations to verify the correctness of the monitor automatically.

The setup of building additional functionality on top of an exisiting RTLola monitor was also used by Baumeister et al. [32] to provide visualistation to the user during monitoring. With this approach, the specification is also extended with additional streams to allow comunication between monitor and the visualistation tool build on top, very much the same as with our approach from this thesis.

Chapter 7

Conclusion

This thesis addresses the challenge of operating a runtime monitor under constrained bandwidth and presents an approach to dynamically allocate the available bandwidth to different inputs depending on the current state of the monitor. The approach empowers the monitor to prioritize and allocate bandwidth to streams deemed most important at any given moment.

To facilitate this, we extended the RTLola syntax to allow the inclusion of scheduling annotations inside the specification. These annotations assist the monitor in determining a schedule of the specific streams to update and their corresponding timing. For this purpose, we gave the annotations a semantic meaning and integrated them into the existing RTLola semantics. Additionally, we developed an algorithm capable of transforming annotated specifications into specifications without annotations, embedding the computed schedule as additional output streams. The resulting specification is then monitored through a regular RTLola backend, while a scheduler interfaces between the backend and sensors. The scheduler is responsible for querying inputs from sensors in accordance with the established schedule.

We have evaluated this approach, employing both a generated trace and simulations with the AirSim drone simulator. The results were promising, demonstrating the effectiveness of the scheduled monitor in detecting violations early on, while consuming significantly less bandwidth compared to a monitor receiving all inputs at a fixed frequency. Beyond bandwidth efficiency, the reduced data intake without compromising monitoring quality translated into lower energy consumption of the monitor, a critical consideration for cyber-physical systems. While acknowledging existing limitations in the current implementation, this approach might be worth further investigation in future work.

7.1. Future Work

In the context of future work, it could be valuable to investigate the performance of this approach with more real-life monitoring settings. Given that sensors frequently automatically generate data, the process of querying data from the sensors imposes additional bandwidth, which is redundant if sensors transmit data independently at a fixed frequency. It would be interesting to examine whether the scheduling approach in such scenarios would still be beneficial.

Regarding the scheduling approach itself, there are opportunities for further development in future work. The algorithm constructing a schedule based on scheduling annotations could be improved to eliminate existing restrictions. Exploring additional kinds of constraints and potentially enabling the mixing of different kinds of constraint values in the same specification would be a worthwhile investigation. Moreover, the exploration of alternative kinds of constraint values may lead to even more intuitive annotations. Extending the scheduling approach to support the spawn and close clauses of the RTLola language would allow to dynamically start and stop scheduling of particular inputs during runtime, thereby preserving additional bandwidth.

On a different note, the scheduler could be integrated into the RTLola compiler, embedding the scheduler together with the monitor directly as code. This integration would enable leveraging the benefits of compiled code as the enhanced performance of the scheduled monitor.

Appendix A

Appendix

A.1. Annotated Geofence Specification

The entire annotated specification for the simplified geofence in Sect. 5.1.1:

→ Sec. 5.1.1, p. 67

```
import math

input a : Float64 schedule max: 4.0s

constant LOWER_BOUND : Float64 := 0.0
constant UPPER_BOUND : Float64 := 5.0

output outside_bounds := a ⩽ LOWER_BOUND ∨ a ⩾ UPPER_BOUND
output move_outside := outside_bounds ∧ ¬
    outside_bounds.offset(by:-1).defaults(to:false)

output distance_to_upper_bound := UPPER_BOUND - a
output distance_to_lower_bound := a - LOWER_BOUND
output distance_to_bound := min(distance_to_upper_bound, distance_to_lower_bound)

output outside_bounds_trigger
  eval @a when distance_to_bound ⩽ 0.6 with move_outside schedule min: 0.8s
  eval @a when distance_to_bound ⩽ 0.8 with move_outside schedule min: 1.5s
  eval @a when distance_to_bound ⩽= 1.2 with move_outside schedule min: 3.0s
  eval @a with move_outside schedule min: 4.0s

trigger outside_bounds_trigger "outside bound"
```

A.2. Translated Geofence Specification

The resulting specification after translating the annotated specification from Appendix A.1 into regular RTLola:

```
import math

input a : Float64

output outside_bounds : Bool
  eval @a with a ⩽ 0.0 ∨ a ⩾ 5.0

output move_outside : Bool
  eval @a with outside_bounds ∧ ¬outside_bounds.offset(by:-1).defaults(to: false)

output distance_to_upper_bound : Float64
  eval @a with 5.0 - a

output distance_to_lower_bound : Float64
  eval @a with a - 0.0

output distance_to_bound : Float64
  eval @a with min(distance_to_upper_bound, distance_to_lower_bound)

output outside_bounds_trigger : Bool
  eval @a with move_outside

output trigger_0 : Bool
  eval @a with outside_bounds_trigger

trigger trigger_0 "outside bound"

output schedule_0 @a :=
    if distance_to_bound ⩽ 0.6 then 0.8
    else if ¬(distance_to_bound ⩽ 0.6) ∧ distance_to_bound ⩽ 0.8 then 1.5
    else if ¬(distance_to_bound ⩽ 0.6) ∧ ¬(distance_to_bound ⩽ 0.8)
       ∧ distance_to_bound ⩽ 1.2 then 3.0
    else if ¬(distance_to_bound ⩽ 0.6) ∧ ¬(distance_to_bound ⩽ 0.8)
       ∧ ¬(distance_to_bound ⩽ 1.2) then 4.0
    else 4.0
```

Note that the schedule stream is represented as a single eval clause employing an if-else-cascade, rather than employing multiple eval clauses. This functionally equivalent adaption is necessary due to the current lack of support for multiple eval clauses in the RTLola interpreter. To accommodate this limitation, we have introduced a command-line flag to the translator, allowing users to opt for this representation until multiple eval clauses are integrated into the interpreter.

A.3. Annotated AirSim Specification

The entire annotated specification for the evaluation with AirSim in Sect. 5.2:

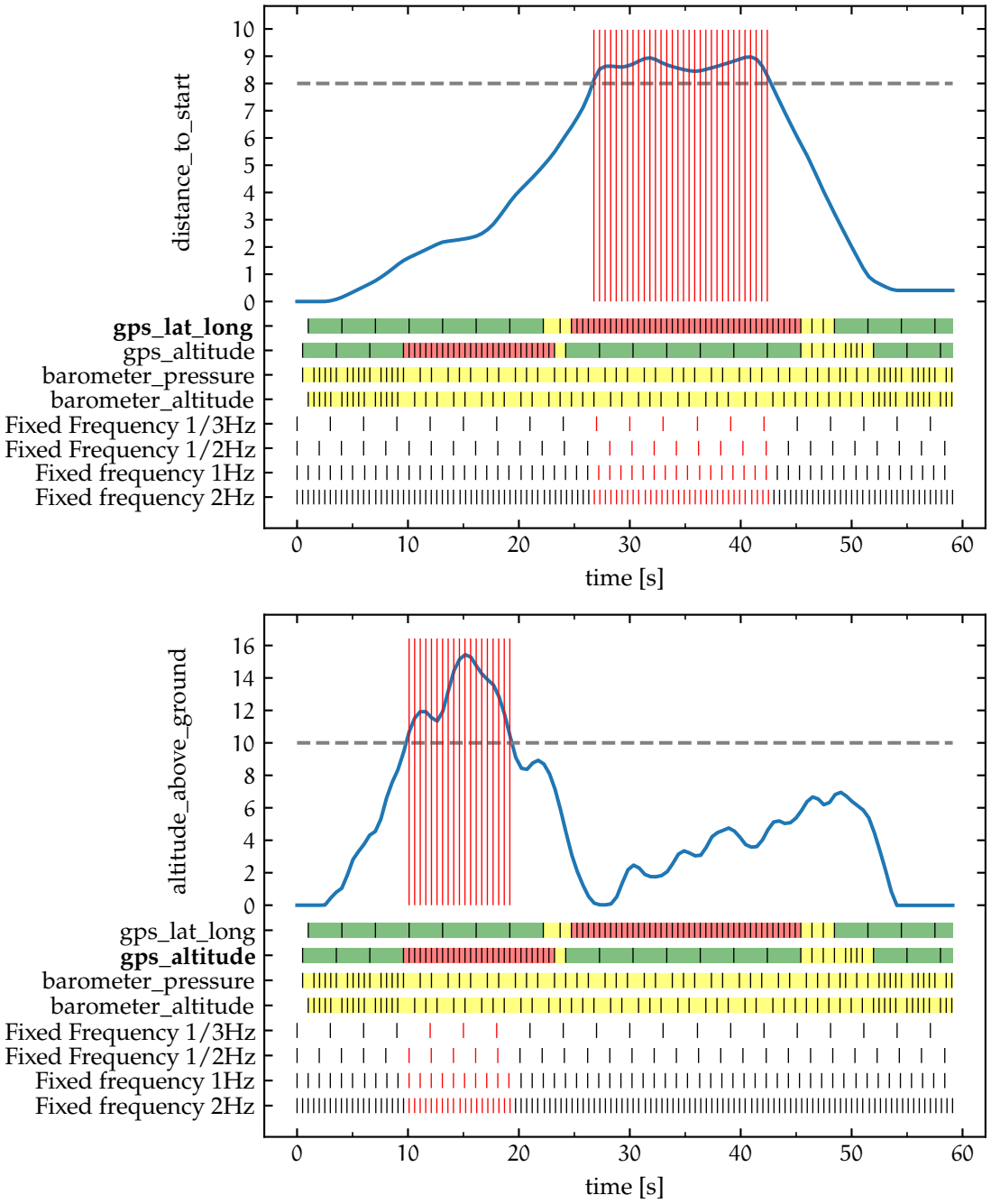
→ Sec. 5.2, p. 72

```
import math
input gps_lat_long : (Float64, Float64) schedule max: high and min: low
input barometer_pressure : Float64 schedule max: medium and min: medium
input barometer_altitude : Float64 schedule max: medium and min: medium
input gps_altitude : Float64 schedule max: high and min: low

output lat := gps_lat_long.0
output long := gps_lat_long.1
output start_lat @gps_lat_long := start_lat.offset(by:-1).defaults(to: lat)
output start_long @gps_lat_long := start_long.offset(by:-1).defaults(to: long)
output start_altitude @gps_altitude := start_altitude.offset(by:-1).defaults(to:
    gps_altitude)

output distance_to_start := sqrt((lat-start_lat)*(lat-start_lat)
   + (long-start_long)*(long-start_long))*10000.0
output altitude_above_ground := gps_altitude - start_altitude

output geofence := distance_to_start ⩾ 8.0

output scheduled_geofence
   eval @gps_lat_long when distance_to_start ⩽ 4.0
                      with geofence schedule min: low
   eval @gps_lat_long when distance_to_start ⩽ 6.0
                      with geofence schedule min: medium
   eval @gps_lat_long with geofence schedule min: high

trigger scheduled_geofence "outside geofence"

output altitude_bound := altitude_above_ground ⩾ 10.0

output scheduled_altitude_bound
   eval @gps_altitude when altitude_above_ground ⩽ 5.0
                      with altitude_bound schedule min: low
   eval @gps_altitude when altitude_above_ground ⩽ 7.5
                      with altitude_bound schedule min: medium
   eval @gps_altitude with altitude_bound schedule min: high

trigger scheduled_altitude_bound "altitude too high"
```

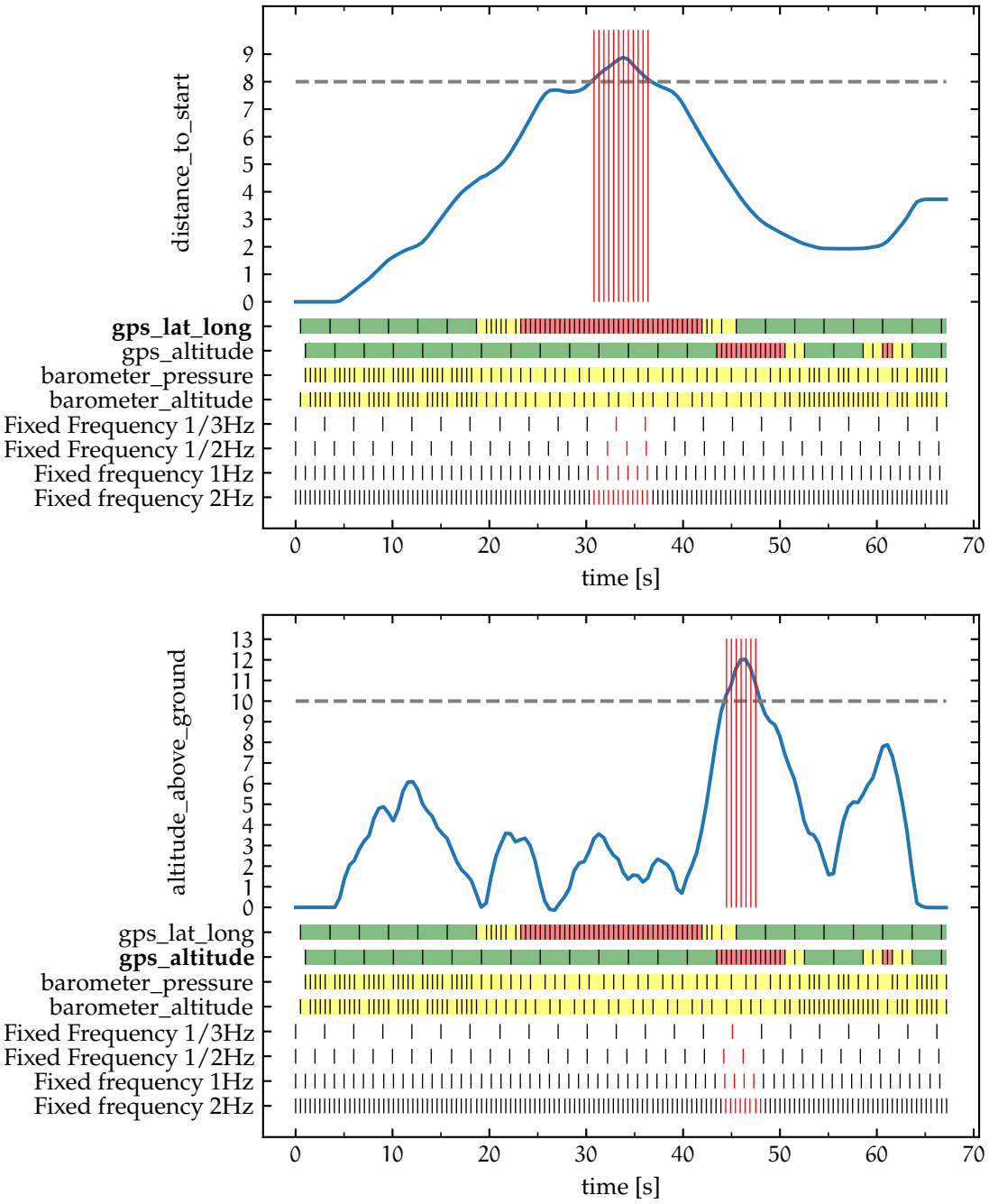
A.4. Additional Simulator Runs

→ Sec. 5.2, p. 72 The additional simulator runs used for the evaluation with AirSim in Sect. 5.2.

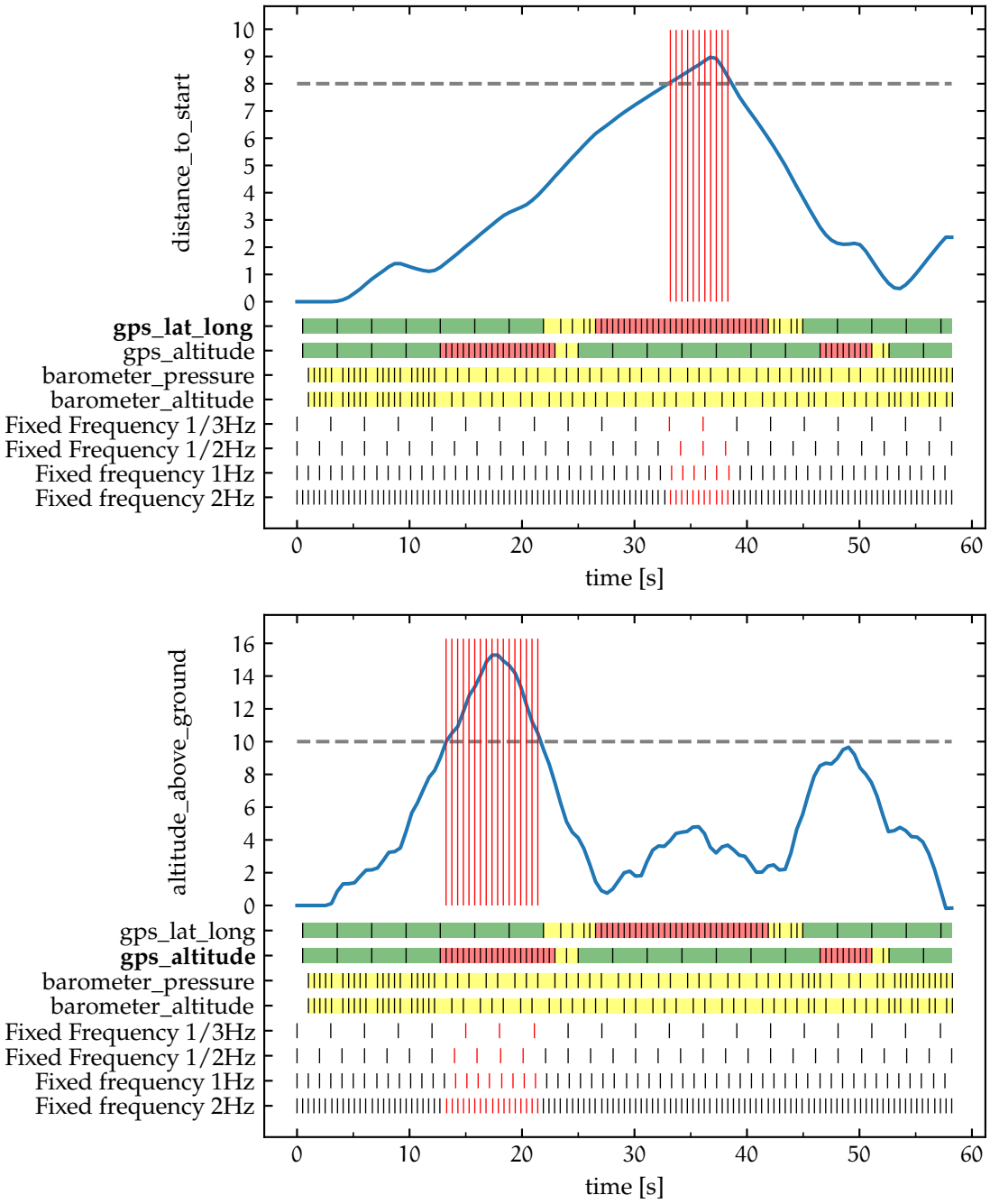
Additional Run 1



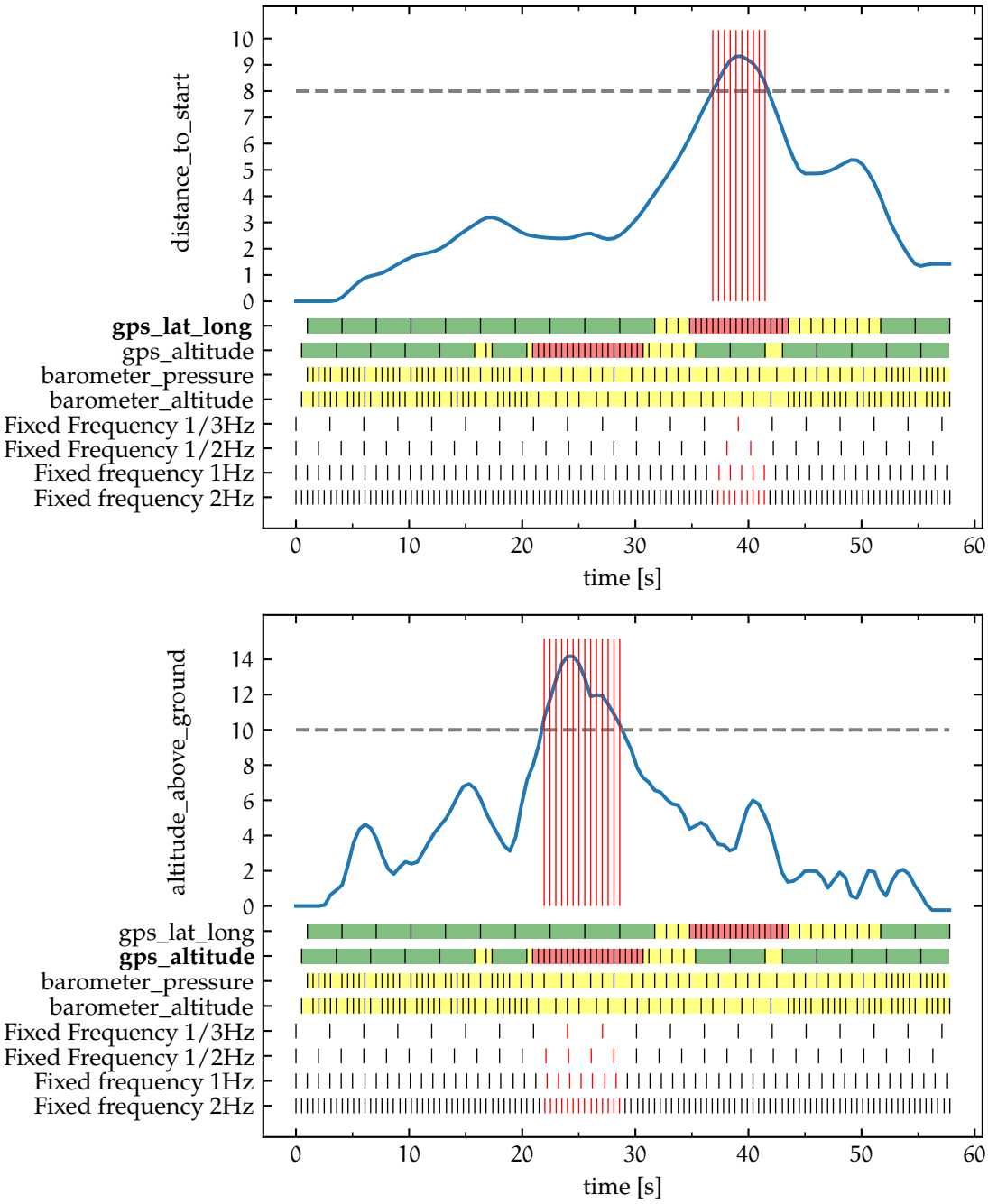
Additional Run 2



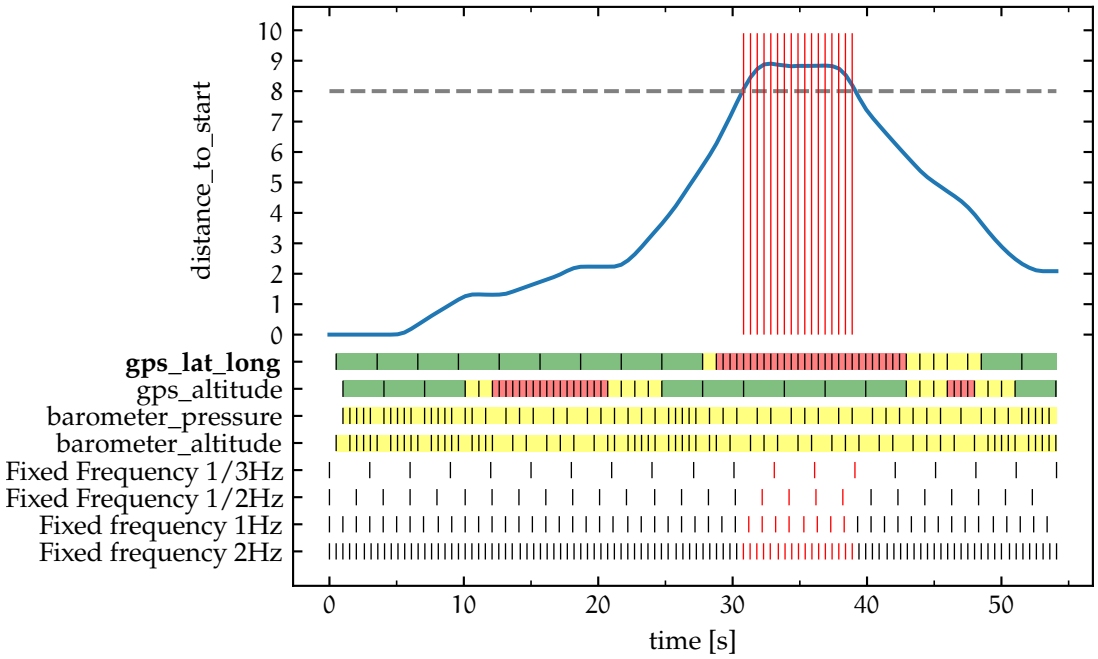
Additional Run 3

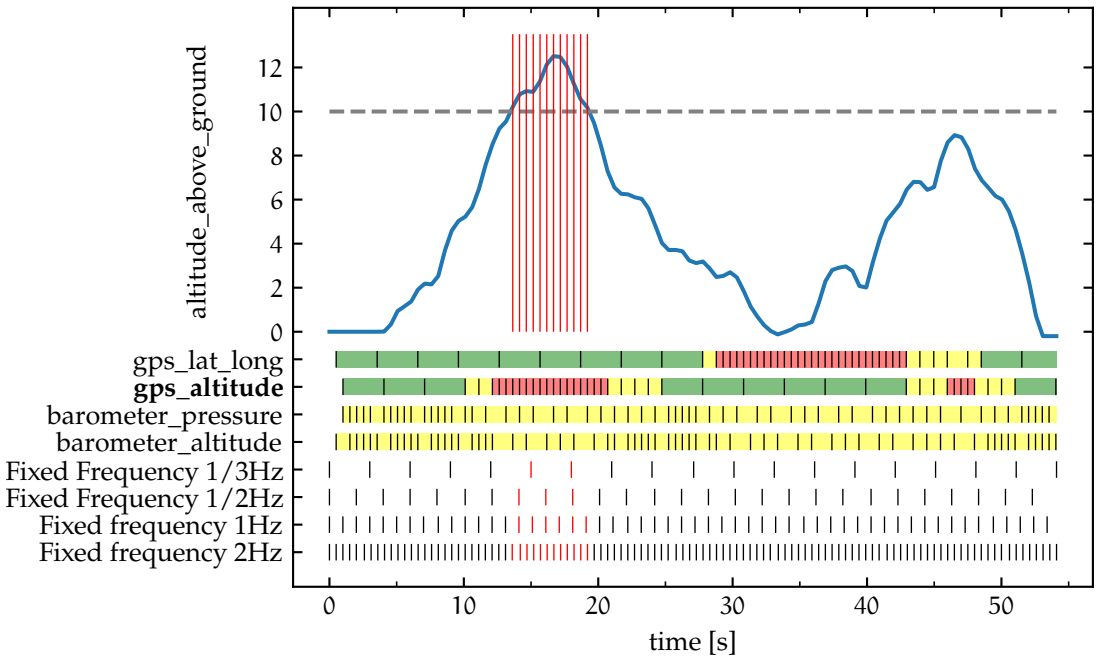


Additional Run 4

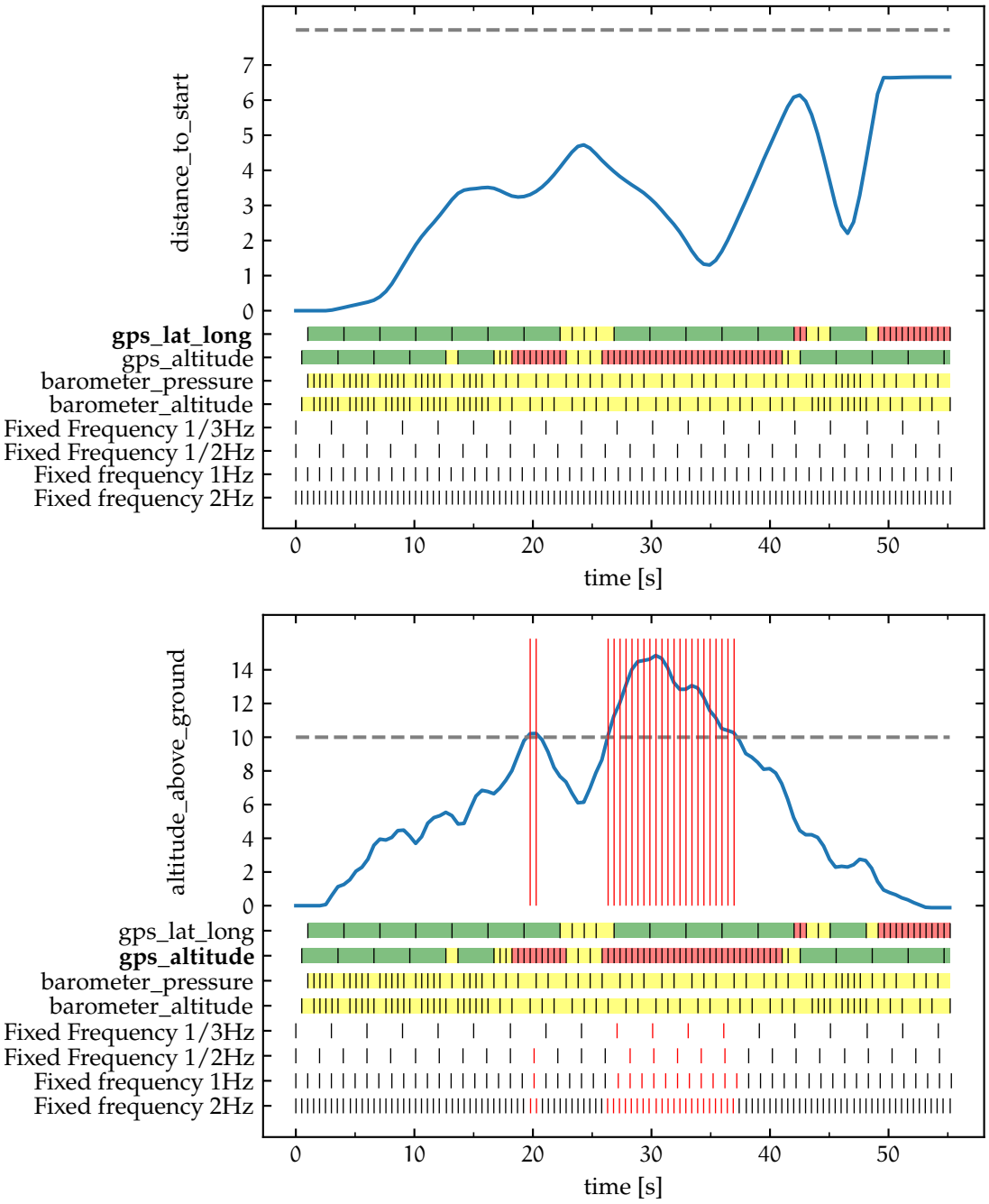


Additional Run 5

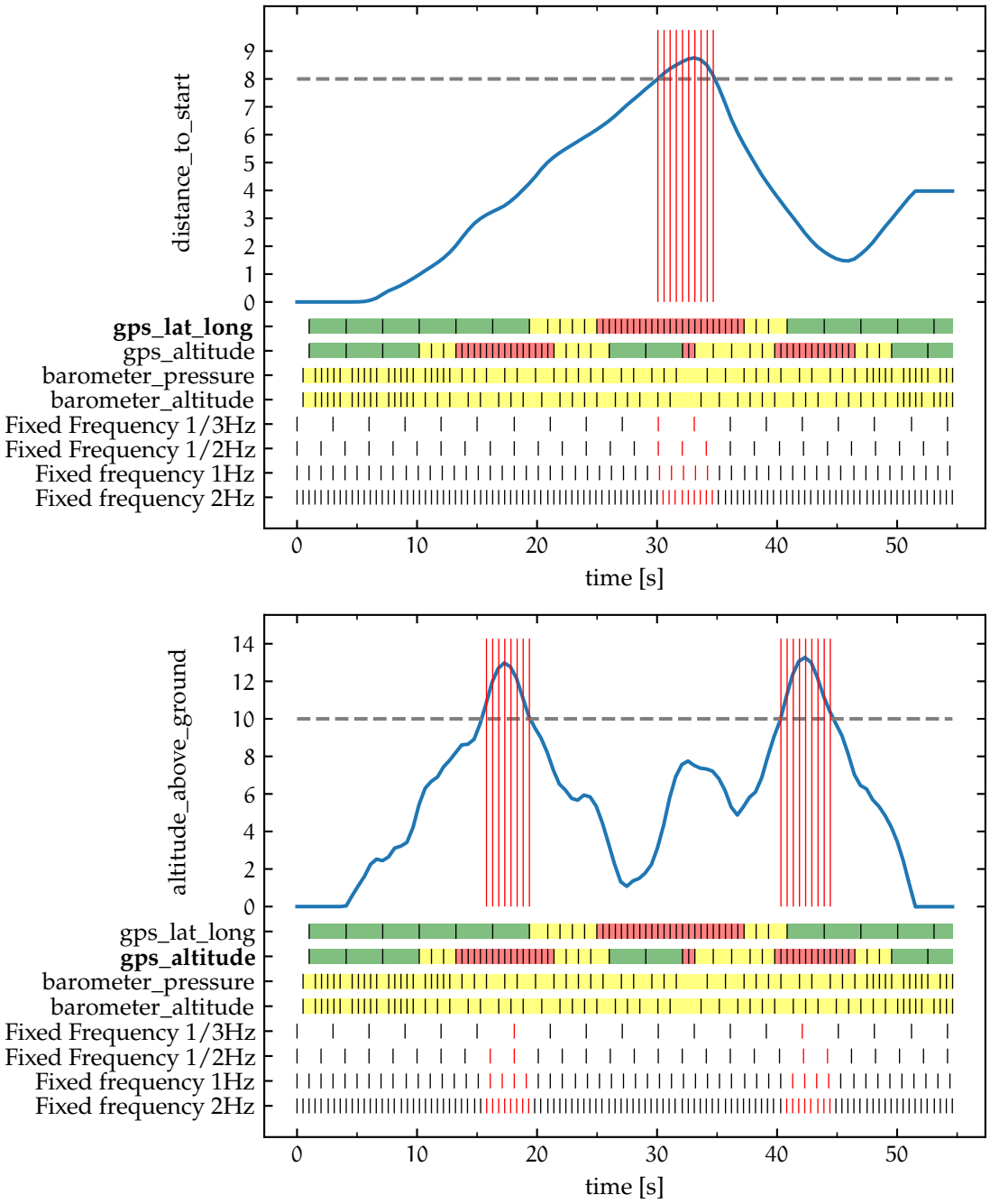




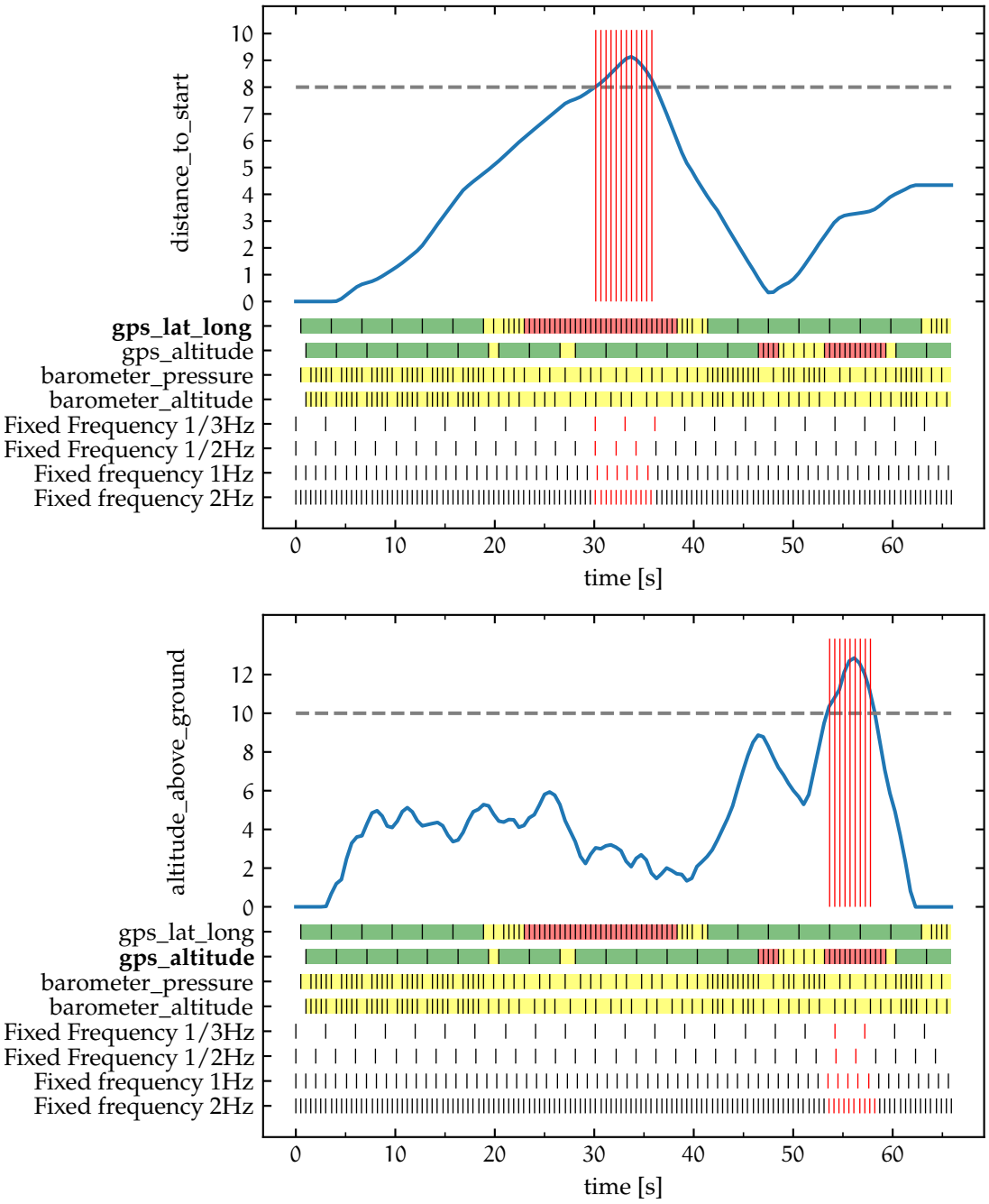
Additional Run 6



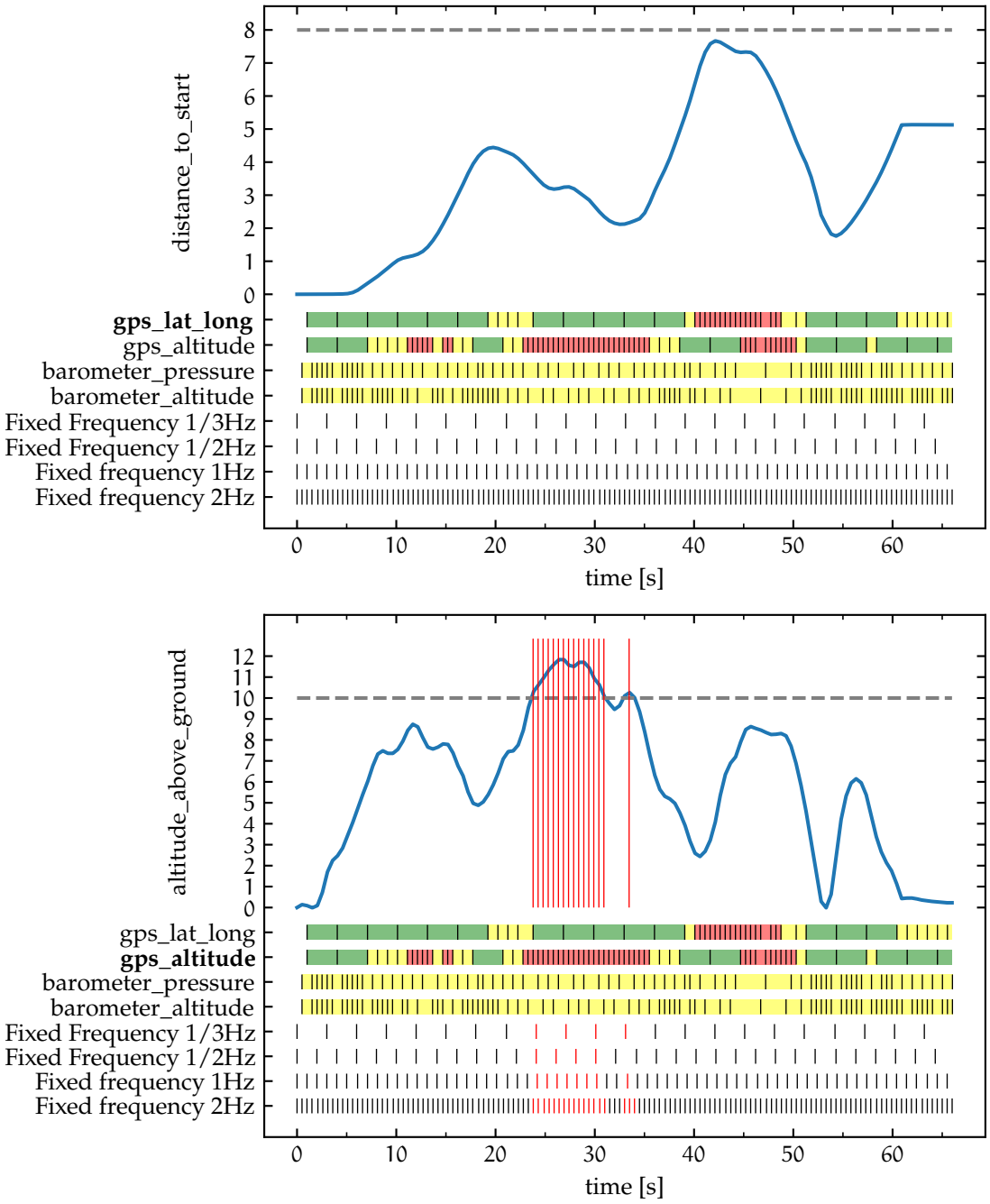
Additional Run 7



Additional Run 8



Additional Run 9



Bibliography

[1] Peter Faymonville et al. "Streamlab: Stream-based monitoring of cyber-physical systems". In: *International Conference on Computer Aided Verification*. Springer. 2019, pp. 421–431.

[2] Ben D'Angelo et al. "Lola: Runtime Monitoring of Synchronous Systems". In: *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*. Burlington, Vermont: IEEE Computer Society Press, June 2005, pp. 166–174.

[3] Maximilian Schwenger. "Statically-analyzed stream monitoring for cyber-physical Systems". PhD thesis. Jan. 2022.

[4] Maximilian Schwenger, Jan Baumeister, and Florian Kohn. *rtlola-frontend*. `https://crates.io/crates/rtlola-frontend`. Version 0.6.1. A frontend for the RTLola runtime verification framework. CISPA - Helmholtz Center for Information Security, 2023. URL: `http://rtlola.org`.

[5] *pest*. `https://crates.io/crates/pest`. Version 2.7.5. The Elegant Parser. pest-parser, 2023. URL: `https://pest.rs`.

[6] The Rust Documentation Team. *The Rust Programming Language - Chapter 19.4 - Using the Newtype Pattern for Type Safety and Abstraction*. The Rust Programming Language. 2023. URL: `https://doc.rust-lang.org/book/ch19-04-advanced-types.html#using-the-newtype-pattern-for-type-safety-and-abstraction`.

[7] Maximilian Schwenger, Jan Baumeister, and Florian Kohn. *rtlola-interpreter*. `https://crates.io/crates/rtlola-interpreter`. Version 0.9.0. An interpreter for RTLola specifications. CISPA - Helmholtz Center for Information Security, 2021. URL: `http://rtlola.org`.

[8] Shital Shah et al. "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles". In: *Field and Service Robotics*. Ed. by Marco Hutter and Roland Siegwart. Cham: Springer International Publishing, 2018, pp. 621–635. ISBN: 978-3-319-67361-5.

[9] *Sensors in AirSim*. Microsoft Research. 2021. URL: `https://microsoft.github.io/AirSim/sensors/`.

[10] Doron Drusinsky. "The temporal rover and the ATG rover". In: *SPIN Model Checking and Software Verification: 7th International SPIN Workshop, Stanford, CA, USA, August 30-September 1, 2000. Proceedings 7*. Springer. 2000, pp. 323–330.

[11] Klaus Havelund and Grigore Roşu. "Monitoring java programs with java pathexplorer". In: *Electronic Notes in Theoretical Computer Science* 55.2 (2001), pp. 200–217.

[12] Nicholas Halbwachs et al. "The synchronous data flow programming language LUSTRE". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320.

[13] Frédéric Boussinot and Robert De Simone. "The ESTEREL language". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1293–1304.

[14] Thierry Gautier, Paul Le Guernic, and Loic Besnard. *Signal: A declarative language for synchronous programming of real-time systems*. Springer, 1987.

[15] Lukas Convent et al. "TeSSLa: temporal stream-based specification language". In: *Formal Methods: Foundations and Applications: 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26–30, 2018, Proceedings 21*. Springer. 2018, pp. 144–162.

[16] Jan Baumeister et al. "RTLola Cleared for Take-Off: Monitoring Autonomous Aircraft". In: *arXiv preprint arXiv:2004.06488* (2020).

[17] Florian Kohn. "A Stream-based Approach to Network Intrusion Detection". Bachelor Thesis. Saarland University, 2019.

[18] Frederik Scheerer. "Monitoring Smart Contracts with RTLola". Bachelor Thesis. Saarland University, 2021.

[19] Marvin Hofman. "Runtime Verification of Critical Web-based Systems with Lola". Bachelor Thesis. Saarland University, 2018.

[20] Sebastian Biewer et al. "On the road with RTLola: Testing real driving emissions on your phone". In: *International Journal on Software Tools for Technology Transfer* 25.2 (2023), pp. 205–218.

[21] Di Xu, Chenggang Wu, and Pen-Chung Yew. "On mitigating memory bandwidth contention through bandwidth-aware scheduling". In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. 2010, pp. 237–248.

[22] Anton V. Eremeev et al. *Multi-Core Processor Scheduling with Respect to Data Bus Bandwidth*. 2020. URL: `http://arxiv.org/abs/2010.16058`.

[23] Sara Afshar et al. "Resource sharing under global scheduling with partial processor bandwidth". In: *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE. 2015, pp. 1–12.

[24] Ankit Agrawal. "Hardware Contention-Aware Real-Time Scheduling on Multi-Core Platforms in Safety-Critical Systems". doctoralthesis. Technische Universität Kaiserslautern, 2019, pp. XVII, 101. URL: http://nbn-resolving.de/urn:nbn:de:hbz:386-kluedo-56124.

[25] Md Zakirul Alam Bhuiyan et al. "Energy and bandwidth-efficient wireless sensor networks for monitoring high-frequency events". In: *2013 IEEE International Conference on Sensing, Communications and Networking (SECON)*. IEEE. 2013, pp. 194–202.

[26] Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. "Sampling-based runtime verification". In: *FM 2011: Formal Methods: 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings 17*. Springer. 2011, pp. 88–102.

[27] Samaneh Navabpour, Borzoo Bonakdarpour, and Sebastian Fischmeister. "Path-aware time-triggered runtime verification". In: *Runtime Verification: Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers 3*. Springer. 2013, pp. 199–213.

[28] Scott D Stoller et al. "Runtime verification with state estimation". In: *Runtime Verification: Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers 2*. Springer. 2012, pp. 193–207.

[29] Xiaowan Huang et al. "Software monitoring with controllable overhead". In: *International Journal on Software Tools for Technology Transfer* 14 (2012), pp. 327–347.

[30] Jan Baumeister et al. "FPGA Stream-Monitoring of Real-Time Properties". In: *ACM Trans. Embed. Comput. Syst.* 18.5s (Oct. 2019). ISSN: 1539-9087. DOI: 10.1145/3358220. URL: https://doi.org/10.1145/3358220.

[31] Bernd Finkbeiner et al. "Verified rust monitors for lola specifications". In: *International Conference on Runtime Verification*. Springer. 2020, pp. 431–450.

[32] Jan Baumeister et al. "Real-Time Visualization of Stream-Based Monitoring Data". In: *Runtime Verification*. Ed. by Thao Dang and Volker Stolz. Cham: Springer International Publishing, 2022, pp. 325–335. ISBN: 978-3-031-17196-3.