Pacing Types: Safe Monitoring of Periodic and Conditional Streams

Saarland University

Department of Computer Science

Master's Thesis

submitted by

Florian Kohn

Saarbrücken, September 2025

Supervisor:	Prof. Bernd Finkbeiner, Ph.D.
Advisor:	Arthur Correnson
Reviewer:	Prof. Bernd Finkbeiner, Ph.D.
	Dr. Hazem Torfah
Submission:	September 30, 2025

Abstract

Runtime monitoring is a safety assurance mechanism that validates a system's behavior against a formal specification at runtime. In stream-based monitoring, a monitor aggregates and transforms input streams, such as sensor readings or system metrics, into output streams that provide temporal or statistical assessments of system health. In practice, input streams may arrive at different rates because sensors operate at different frequencies, requiring the monitor to correctly handle asynchronous data arrival. Frameworks such as RTLola address this through pacing annotations, which specify precisely when streams must be evaluated.

Since the monitor is part of a safety-critical system, preventing runtime errors is essential. However, inconsistent pacing annotations can introduce subtle errors that result in attempts to access unavailable stream values.

This thesis extends RTLola's type-based analysis to support periodic and conditionally evaluated streams, both of which are needed to specify complex real-time behavior. Conditionally evaluated streams use *when* clauses to refine evaluation time points based on boolean conditions. This allows, for example, the exclusion of time points where a division by zero would occur. Periodic streams emit values at fixed intervals, enabling the monitor to compute system verdicts independently of input arrival, such as for detecting deadlocks or failed sensors. We encode periodically paced streams as a special case of *when* clauses and prove type-system soundness with respect to RTLola's relational semantics. This guarantees that pacing annotations are consistent such that specifications can be safely monitored without invalid data accesses.

Acknowledgements

First and foremost, I would like to thank Prof. Bernd Finkbeiner for giving me the opportunity to work on a topic of my choice and for his guidance throughout the development of this thesis. I am also very grateful to my colleagues, whose advice (and countless cups of coffee) helped me bring this work to completion.

A special thanks goes to my advisor, Arthur Correnson, for his time, patience, and valuable feedback. I also wish to thank Prof. Bernd Finkbeiner and Dr. Hazem Torfah for reviewing this thesis.

Finally, I would like to express my heartfelt gratitude to my family and friends for their continuous support and encouragement during the creation of this thesis.

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, 30 September, 2025

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, 30 September, 2025

Contents

Chapter 1

Introduction

Cyber-physical systems are safety-critical systems at the boundary of the physical and digital world. They range from power plants to self-driving cars and autonomous drones. Runtime verification is a safety assurance mechanism that observes the behavior of a system at runtime and validates it against a predefined set of rules describing known bad behavior. In runtime monitoring, a so-called monitor is automatically derived from a formal safety specification.

In stream-based runtime monitoring, as visualized in Fig. 1.1, the monitor captures the observations of the system through input streams representing a possible infinite sequence of (sensor) values. Output streams transform and aggregate input and other output streams through stream expressions to derive an assessment of the system's health. Special output streams, called triggers, notify the system's operator about the system's state based on boolean conditions. In practice, observations of the system stem from a variety of sensors of the system. Different sensors usually produce values at different frequencies. As a result, the monitor must handle the asynchronous arrival of input data. High-level specification frameworks such as RTLola provide fine-grained

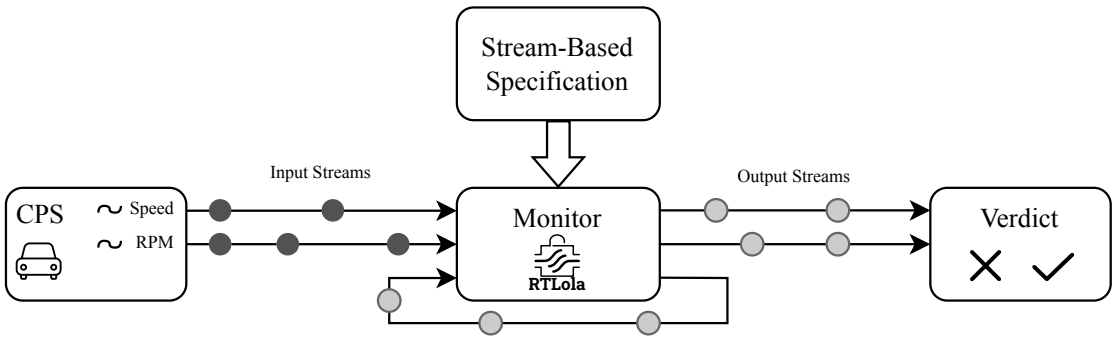


Figure 1.1.: A High-Level Overview of Stream-based Monitoring

synchronization primitives called pacing annotations for this. While significantly easing the specification of monitors, inconsistent pacing annotations are also a source of subtle errors, such as requiring the monitor to access unavailable stream values.

As monitors become part of the cyber-physical system, they also become safety-critical. Hence, ensuring that they are free of runtime errors such as those stemming from inconsistent pacing annotations is of uttermost importance. Recently, a type-based analysis for the consistency of pacing annotations has been explored [Koh+25]. However, the presented analysis lacks the real-time and conditional evaluation features of RTLola. Yet, it was shown that these features are crucial for monitoring real-world systems [Bau+24b].

The real-time features of RTLola include periodic pacing annotations, which declare that a stream must evaluate at a fixed frequency. These are essential for the monitor to proactively produce verdicts such as detecting sensor failures, which is impossible, if the monitor can only react to its inputs as it cannot react to non-existing data. Conditional evaluation adds *when* clauses to streams that refine their evaluation time points at runtime based on a boolean condition. These capture a variety of use cases such as restricting the evaluation of a stream to the time points where the divisor of a division is unequal zero.

This thesis extends the existing type based analysis of pacing annotations in RTLola with *when* conditions. For this, we provide a relational semantics for RTLola with *when* conditions and show that the extended type system is sound, meaning that type safety proofs the absence of runtime errors, using logical relations. We demonstrate how the real-time features of RTLola can be encoded with *when* conditions and present a case study analyzing the capabilities and limitations of the presented extension. As an objective measure we compare the performance of a native implementation of the real-time features of RTLola against their encoded pendants.

1.1. Motivation

To motivate stream-based monitoring and the benefit of pacing annotations, consider the following simple RTLola specification monitoring the velocity of a car:

```
1 input position: Float
2
3 output velocity @position := (position - position.offset(by: -1).defaults(to:
      position)) * 36.0
4
5 trigger @position velocity > 150 "Driving too fast"
```

The `input` stream declared in line one of the specification captures the current position the car as a floating point number. To compute the velocity of the car, the `output` stream `velocity` computes the difference between the current position and the previous one. The previous position is obtained using an `offset` access to the `position` stream, with the `default` operator providing a fallback value when no previous value exists. Assuming the car's position is sampled approximately every 100ms, the resulting velocity is scaled to kilometers per hour by multiplying with the constant factor 36. Finally, the `trigger` in line five compares the velocity to the maximum permitted speed. Both the trigger and the `velocity` stream are updated whenever the `position` input receives a new value, as specified by the pacing annotation `@position`.

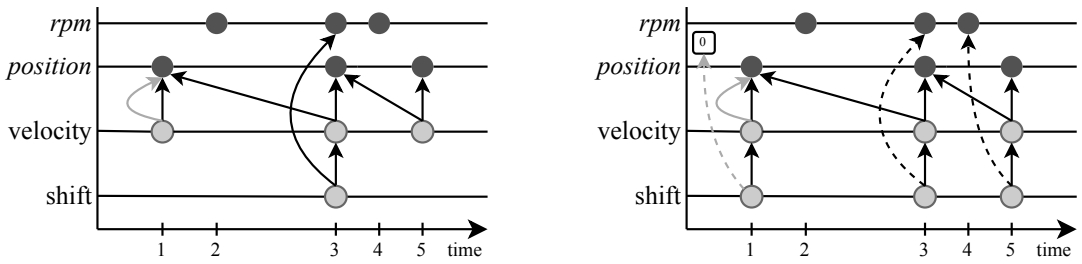
1.1.1. Asynchrony

In practice, the monitor must observe multiple sensors to provide meaningful verdicts about the system's health. To illustrate this, we extend the example with an additional `rpm` input stream that records the engine's revolutions per minute.

```
1 input rpm: Float
2 input position: Float
3
4 output velocity @position := (position - position.offset(by: -1).defaults(to:
      position)) * 36.0
5
6 trigger @position velocity > 150.0 "Driving too fast"
7
8 output shift @position && rpm := velocity < 30.0 && rpm.hold(or: 0.0) > 3000.0
9 trigger @position && rpm shift "Shift to higher gear"
```

Specification 1.1: An RTLola Specification with two Input Streams.

The RPM values allow the monitor to compute more elaborate verdicts. For instance, the `shift` stream indicates that the driver should switch to a higher gear if the engine's RPM is excessive while the vehicle speed is low. Unlike before, the pacing annotation now requires both the `position` and `rpm` streams. Consequently, the `shift` stream evaluates only when new position and RPM values arrive simultaneously. This behavior is visualized in Fig. 1.2a. There, dots symbolize the presence of stream values while arrows



(a) Timing Diagram for Spec. 1.1.

(b) Timing Diagram for Spec. 1.2.

Figure 1.2.: Timing Diagrams for the Specification in Spec. 1.1 and Spec. 1.2.

show the dependency between these values. Light-grey arrows represent dependencies stemming from default values.

In reality, however, different sensors produce values at different pace which can cause the stream and with that the trigger to not detect critical system states. This can be observed in Fig. 1.2a, where the shift stream does not evaluate at time points one, two, four and five, because the rpm and position inputs do not coincide.

The Hold Operator. To address such asynchrony, RTLola provides the `hold` operator. Unlike synchronous stream accesses, which refer to a stream's value in the current cycle, the hold operator refers to its most recent value, which may originate from an earlier time point. If no previous value exists, a default must be provided, similar to the `offset` operator. With the hold operator, the above specification is modified as follows:

```
1 input rpm: Float
2 input position: Float
3
4 output velocity @position := (position - position.offset(by: -1).defaults(to:
      position)) * 36.0
5
6 trigger @position velocity > 150.0 "Driving too fast"
7
8 output shift @position := velocity < 30.0 && rpm.hold(or: 0.0) > 3000.0
9 trigger @position shift "Shift to higher gear"
```

Specification 1.2: An RTLola Specification featuring the Hold Operator.

The `shift` stream now accesses the RPM values via the hold operator. This removes the requirement that an RPM measurement must be present in the current evaluation cycle. Accordingly, its pacing annotation is adapted so that the stream evaluates whenever a new position is received. This ensures that the issue illustrated above is avoided. In Fig. 1.2b, dashed arrows denote dependencies from hold accesses. Compared to Fig. 1.2a, the `shift` stream now also evaluates at time points one and five. However,

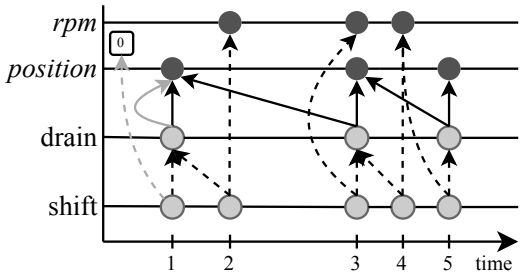


Figure 1.3.: A Timing Diagram for the Specification in Spec. 1.3.

pacing annotations serve purposes beyond merely capturing dependencies between streams.

Disjunctive Pacing Annotations A monitor synthesized from the above specification may still miss critical events, such as high RPM values without vehicle movement (e.g., when the engine is revved while stationary). To address this, the monitor should evaluate the `shift` stream whenever either a position or an RPM reading is received. In RTLola this behavior is expressed using disjunctive pacing types:

```
1 input rpm: Float
2 input position: Float
3
4 output velocity @position := (position - position.offset(by: -1).defaults(to:
      position)) * 36.0
5
6 trigger @position velocity > 150.0 "Driving too fast"
7
8 output shift @position || rpm := velocity.hold(or: 0.0) < 30.0 && rpm.hold(or: 0.0) >
      3000.0
9 trigger @position || rpm shift "Shift to higher gear"
```

Specification 1.3: An RTLola Specification with disjunctive Pacing Annotations

In this specification, the pacing annotation of the `shift` stream explicitly encodes this disjunction. Unlike before, the annotation no longer requires the presence of a position value, allowing the stream to evaluate when only an RPM measurement is received. Accordingly, the access to the position stream in `shift` is changed to be asynchronous. Fig. 1.3 illustrates this behavior: compared to Fig. 1.2b, the `shift` stream now also evaluates at time points two and four.

Summary In summary, RTLola provides three methods to access stream values:

1. **Direct Synchronous Access.** Refers to the current value of a stream by its name.

2. **Offset Synchronous Access.** Refers to a past value of a stream using the `offset` operator with a `default` fallback, but requires the stream to have a value in the current cycle. This guarantees freshness of the accessed past value.

3. **Asynchronous Access.** Refers to the most recent value of a stream using the `hold` operator, regardless of whether it was produced in the current or a past cycle. If no past value exists, a default must be specified. Unlike the offset access, hold imposes no bound on the age of the value.

Stream evaluation is further controlled by **pacing annotations**, written after the stream name and preceded by `@`. These are positive boolean formulas over input streams, where each atomic proposition evaluates to `true` whenever the corresponding input stream receives a value.

1.1.2. Handling Sensor Failure

A limitation of *event-based* pacing annotations is that they only enable the monitor to react to input events. If the system becomes unresponsive and no measurements are produced, the monitor cannot detect this absence. To address this, RTLola supports periodic pacing annotations, which specify a fixed frequency for stream evaluation. For example, assuming RPM measurements arrive at 1Hz, the specification can be extended to detect when this assumption is violated.

```
1  input position: Float
2  input rpm: Float
3
4  output velocity @position := (position - position.offset(by: -1).defaults(to:
       position)) * 36.0
5  trigger @position velocity > 150.0 "Driving too fast"
6
7
8  output count @1Hz := rpm.aggregate(over: 1min, using: count)
9  trigger @1Hz count < 60 "RPM Sensor Failure"
10
11 output shift @position || rpm := velocity.hold(or: 0.0) < 30.0 && rpm.hold(or: 0.0) >
       3000.0
12 trigger @position || rpm shift "Shift to higher gear"
```

Specification 1.4: A Specification that detects Sensor Failures using Periodic Streams.

The additional `count` stream evaluates once per second and uses a sliding window aggregation to count the number of RPM values received in the past minute. The trigger in line nine asserts that this count is at least 60. The sliding window aggregation is expressed using the `aggregate` function, which takes two named arguments: `over`, specifying the window duration, and `using`, defining the aggregation function applied to all values of the target stream within that duration.

Increasing Robustness. Sensor measurements are often noisy and may fluctuate around the true physical value. To account for this, we define the `sum` stream, which sums all RPM values over a one-minute sliding window. Dividing this sum by the `count` yields the average RPM over the last minute. The `shift` stream is then adapted to use this average.

```
1  input position: Float
2  input rpm: Float
3
4  output velocity @position := (position - position.offset(by: -1).defaults(to:
       position)) * 36.0
5  trigger @position velocity > 150.0 "Driving too fast"
6
7
8  output count @1Hz := rpm.aggregate(over: 1min, using: count)
9  output sum @1Hz := rpm.aggregate(over: 1min, using: sum)
10
11 output avg_rpm @1Hz := sum / count
12 trigger @1Hz count < 60
13
14 output shift @position || rpm := velocity.hold(or: 0.0) < 30.0 && avg_rpm.hold(or:
       0.0) > 3000.0
15 trigger @position || rpm shift "Shift to higher gear"
```

However, static timing information (input events or fixed periods) is insufficient to guarantee safe evaluation. Runtime conditions must also be considered to prevent errors such as division by zero.

1.1.3. Computation Shielding

For example, if the RPM sensor fails to produce values for one minute, the `count` stream evaluates to zero, leading to a division by zero in `avg_rpm`. To avoid such failures, RTLola provides **evaluation conditions**. These are boolean stream expressions that refine the static timing of a stream, ensuring it only evaluates when both the pacing annotation is satisfied and the condition holds. Finally, the above specification is adapted as follows:

```
1  input rpm: Float
2  input position: Float
3
4  output velocity @position := (position - position.offset(by: -1).defaults(to:
       position)) * 36.0
5  trigger @position velocity > 150.0 "Driving too fast"
6
7
8  output count @1Hz := rpm.aggregate(over: 1min, using: count)
9  output sum @1Hz := rpm.aggregate(over: 1min, using: sum)
10
11 output avg_rpm @1Hz when count > 0 := sum / count
12 trigger @1Hz count < 60
```

```
13
14 output shift @position || rpm := velocity.hold(or: 0.0) < 30.0 && avg_rpm.hold(or:
       0.0) > 3000.0
15 trigger @position || rpm shift "Shift to higher gear"
```

The stream `avg_rpm` is now guarded by an evaluation, or *when*, condition specified after the `when` keyword that asserts that the `count` stream must be greater than zero for the `avg_rpm` stream to evaluate.

1.1.4. Problem Statement

The preceding examples demonstrate that expressive pacing annotations and evaluation conditions are essential for monitoring complex cyber-physical systems. At the same time, their expressiveness makes inconsistencies easier to introduce. Consider, for example, the following specification with inconsistent pacing annotations:

```
1 input a: Int
2 input b: Int
3
4 output c @a || b := a + b
```

Specification 1.5: A Specification with faulty Pacing Annotations.

Consider time point three in the timing diagram shown in Fig. 1.4a. In this scenario, it occurs that the stream `a` receives a value at a time point where stream `b` does not. This could lead to the failure of the monitor, as it tries to access the non-existent value when computing the value for stream `c`. This is visualized by the red arrow and red dashed circle in Fig. 1.4a. Such inconsistencies become more subtle with advanced features such as periodic pacing and evaluation conditions.

```
1 output a @2Hz := 42
2 output b @4Hz := a
```

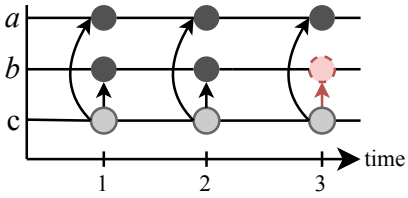
Specification 1.6: A Specification with inconsistent Periodic Pacing Annotations.

Fig. 1.4b shows an inconsistency arising when `b` evaluates twice as often as `a`, requiring nonexistent values of `a` at intermediate time points. In this example, the monitor requires the non-existent values of stream `a` at time points $0.25s$ and $0.75s$ to compute stream `b`. Similarly, Fig. 1.4c illustrates how an evaluation condition can lead to missing values in the following specification:

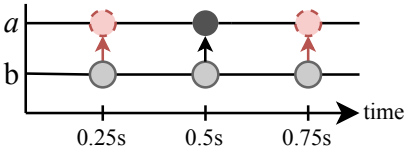
```
1 input a: Int
2 output b @a when a > 10 := a
3 output c @a := b
```

Specification 1.7: A RTLola Specification with inconsistent Evaluation Conditions.

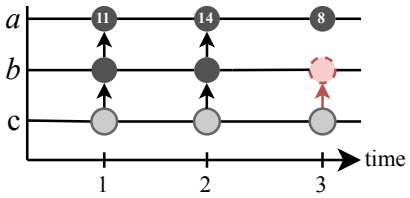
As illustrated, the issue arises when `a` falls below 10, as `b` does not evaluate, while `c` depends on it. An example timing diagram is presented in Fig. 1.4c where the value of `a` is given in white font in the circles.



(a) A Timing Diagram for the Specification in Spec. 1.5.



(b) A Timing Diagram for the Specification in Spec. 1.6.



(c) A Timing Diagram for the Specification in Spec. 1.7.

Figure 1.4.: Timing Diagrams showcasing the inconsistent Pacing Annotations in Spec. 1.5, Spec. 1.6, and Spec. 1.7.

The Problem. To detect such inconsistencies, RTLola employs a type system that reasons about pacing annotations. The central challenge is to formally define this type system and RTLolas semantics such that it can be proven that well-typed specifications are free of runtime errors. Previous work established this only for event-based streams without conditional evaluation.

Contribution. This thesis extends the previous formalization of the type system to cover periodic pacing annotations and evaluation conditions. For this, we first prove type-safety for conditional streams using new type inference rules and then present a translation of periodic annotations and sliding windows into specifications that use only event-based annotations and evaluation conditions. We prove the soundness of the extension to conditional streams by adapting the previous logical relations, showing that well-typed specifications cannot fail at runtime. Finally, we evaluate the performance of translated real-time features against their native implementation in RTLola, thereby quantifying the runtime cost of soundness guarantees.

1.2. Related Work

Runtime Verification and Stream-based Monitoring Runtime verification is a well-established research field that originated in 1999 [Kim+99]. Early approaches relied on temporal logics, such as linear temporal logic [Pnu77; BLS11] and metric temporal logic [Koy90; BKZ17], to specify monitor behavior. Applications span a wide range of systems, including Markov Decision Processes [JTS21; Hen+23], software systems [Jin+12; Sch21], and complex cyber-physical systems [Bau+24b; Bau+20b; Bau+24a].

Complex system properties, such as algorithmic fairness [Bau+25] require expressive specification mechanism that go beyond temporal properties. A prominent example are stream-based specification languages as introduced by Finkbeiner et al. in 2005. [dAn+05]. Thereafter came its successor RTLola [Fay+19; Bau+24a] and other stream-based specification languages such as Tessla [Kal+22], Striver [GS21] or Copilot [PGD24], all of which provide close to fully featured programming languages for specification.

Similar to programming languages, all the above stream-based languages feature a type system that reasons about the data consistency of specifications. In general, type systems as pioneered by Hindley and Milner in 1969 [Hin69; Mil78] are automatic program analyses that guarantee well-defined meaning for programs and specifications. [Pie02, p.208] Copilot [PGD24] and Striver [GS21] both don't feature their own language frontend, but are an embedded domain specific language in the Haskell programming language. TeSSla [Kal+22] and RTLola [Bau+24a] both feature their own language frontend including a type analysis for the validity of data types.

A key distinction among stream-based languages lies in their treatment of asynchronous data. Some languages such as Lola [dAn+05] and Copilot [PGD24] are purely synchronous and do not support asynchronous input. In TeSSla [Kal+22] input streams are modeled as piecewise constant signals, comparable in expressivity to RTLola without synchronous accesses. The Striver [GS21] introduces ticking expressions, conceptually similar to pacing annotations in RTLola. However, it lacks consistency checks between ticking and stream expressions, meaning evaluation is not guaranteed when a tick occurs.

Synchronous Programming and Clocks A related body of research addresses similar challenges in synchronous programming languages, which ensure that values are available when required for computation. Languages such as Lustre [Cas+87], Esterel [BC84], and Signal [GL87] emerged in the 1980s and later inspired languages such as Zélus [Ben+11] and Lucid Synchrone [DGP08]. These languages provide abstractions for reactive systems, analogous to how imperative languages abstract digital processors. Like pacing types in RTLola, synchronous languages employ *clocks* [GL87; CP03; MPP10], which define computation timing, combined with static analyses to de-

tect timing inconsistencies. For instance, [Ben+14] introduces a type system for detecting *non-causal* programs, a property comparable to ensuring a unique evaluation model for a RTLola specification. Work on Lustre [Hal+91] and Lucid Synchrone [CP03] also investigates clock sub-sampling based on boolean conditions and the resulting challenges for timing consistency. The notion of conditional streams in RTLola is directly related to this sub-sampling approach. However, synchronous languages typically address sub-sampling using dependent type theory, whereas the approach in this thesis avoids such requirements. Beyond ensuring clock consistency, recent work has advanced type systems to verify rich temporal properties of signals. For example, [Che+24] presents a refinement type system for the synchronous language MARVeLus. This system reasons about annotations based on LTL [Pnu77], capturing the temporal behavior of signal values.

Chapter 2

Background

This chapter first provides an overview of RTLolas real-time features and conditional streams, for which type safety will be established in this thesis. It then introduces Logical Relations, a proof technique for showing type safety. Finally, it summarizes existing work on formalizing RTLola's type system using the simplified core language StreamCore presented in [Koh+25].

2.1. RTLola

Beyond the features discussed in Sect. 1.1, RTLola as presented in [Bau+24a] includes additional capabilities such as dynamic and parameterized streams. These are useful for monitoring unbounded environments, for example, an unbounded number of aerial vehicles in an airspace. However, ensuring timing consistency between parameterized streams requires reasoning about the existence of stream instances, which lies beyond the scope of this thesis. In addition, RTLola supports conditional streams through *when* conditions in the evaluation clause.

2.1.1. Conditional Streams

In RTLola, a conditional stream such as `avg_rpm` in Sect. 1.1.3 is declared as follows:

```
1 output <name> eval @<pacing> when <condition> with <expression>
```

Semantically, this defines a stream that evaluates to the value of its expression at all positions where both the pacing holds and the condition is true. At positions where the pacing does not hold or the condition evaluates to false, the stream may take any value or remain undefined.

This mechanism enables optimizations in monitor implementations, since no value needs to be computed when the condition is false. For example, inexpensive conditions

can be used to guard more costly monitoring operations, thereby improving performance. Further optimizations of this kind are discussed in [Bau+20a].

Pacing Annotations. Pacing annotations in RTLola have two forms. They may either define an event as a positive boolean formula over input stream variables, or specify a fixed frequency at which a stream must produce a value according to its expression. An event-based annotation holds when the boolean formula evaluates to true, where each input stream variable indicates whether it received a new value in the current evaluation cycle. Periodic pacings, first introduced in [Fay+19], hold according to the fixed frequency relative to the monitors start.

2.1.2. Periodic Streams

Periodic streams are output streams whose pacing annotation specifies a fixed frequency. As discussed in Sect. 1.1, periodic streams extend the monitor from being purely reactive, based on event-driven pacings, to also performing proactive computations, such as detecting sensor failures.

Moreover, periodic streams may include sliding window operations that aggregate all values of a stream within a specified time window. In contrast, sliding window operations are not permitted in event-based streams, as they can lead to unbounded memory requirements [Fay+19].

Sliding Windows Sliding windows apply an n-ary aggregation function f over a value domain V, defined as

$$f \in \mathbb{P}(V) \to V$$

to all values of a stream within a specified time frame d, referred to as the *duration* of the sliding window. Formally, if $V := (v_1, t_1), \ldots, (v_k, t_k)$ denotes the set of timestamped values of a stream, a sliding window operation over that stream with duration d and aggregation function f evaluates at time t as follows:

$$v_r = f(\{v_i \mid (v_i, t_i) \in S \wedge t - d \leqslant t_i < t\})$$

Consider the following example specification of a sliding window operation in RTLola:

```
1 input a: Int
2 output b @2Hz := a.aggregate(over: 1s, using: sum)
```

Specification 2.1: An example sliding window operation.

The sliding window specified in stream `b` states that every 0.5s, `b` equals the sum of all values of stream `a` from the preceding second. Thus, the window duration is 1s, and its aggregation function is the n-ary addition. The semantics of this operation are illustrated in Fig. 2.1.

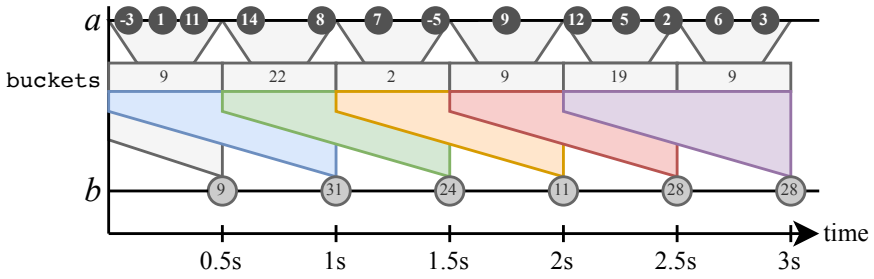


Figure 2.1.: The Semantics of a Sliding Window Operation exemplified.

The buckets shown in the figure represent an implementation strategy for sliding windows that enables bounded-memory monitoring. This is achieved by pre-aggregating the stream values into buckets whose length matches the evaluation period of the aggregation. If the window aggregation is evaluated every x seconds over a duration d, only d/x buckets need to be stored at runtime. This strategy is later used to encode sliding window aggregations in the extended fragment of RTLola by declaring a stream for each required bucket.

2.2. Logical Relations

Logical relations [Sta85] are a proof technique commonly used to establish that a well-typed program satisfies a certain property called type safety. In this thesis, we apply logical relations to show that a specification is safe to monitor. We define this property as the guarantee that, regardless of the values or arrival times of input streams, all streams can always be evaluated to a defined value at runtime. For stream-based languages, and for RTLola in particular, traditional approaches to type safety, such as using progress and preservation lemmas, are not well suited, as they work best for operational semantics rather than the relational semantics used in stream-based languages.

In general, type systems aim to establish semantic properties of programs from their syntactic structure. A type system assigns types to syntactic constructs and imposes constraints on these assignments to ensure type safety. By contrast, logical relations connect types to semantic entities such as value domains or streams of values.

Proofs of type safety using logical relations typically follow a standard structure. First, one defines the logical relations, that is, the semantic interpretation of types and typing contexts. Next, the type safety property is reformulated in terms of these relations; let us call this reformulation *semantic safety*. The proof then proceeds by showing that syntactic well-typedness implies semantic safety, and finally that semantic safety implies the original type safety property. Thus, semantic safety constitutes a strengthening of type safety.

2.2.1. An Introductory Example

Consider the following simplified example from [Sko19], which demonstrates that the strongly typed lambda calculus is strongly normalizing, i.e., all well-typed programs terminate. Assume there are only two types in the language: booleans and unary functions. A first logical relation $\mathcal{V}$ assigns each type its associated set of values.

$$\mathcal{V}[bool] = \{true, false\}$$
$$\mathcal{V}[\tau_1 \to \tau_2] = \{\lambda x : \tau_1 . e \mid \forall v \in \mathcal{V}[\tau_1] . e[v/x] \in \mathcal{E}[\tau_2]\}$$

Similarly, a type is associated with all expressions of that type in the relate $\mathcal{E}$:

$$\mathcal{E}[\tau] = \{e \mid \forall e' . e \rightarrowtail^* e' \land irred(e') \implies e' \in \mathcal{V}[\tau]\}$$

Let $irred(e) \mathrel{\hat{=}} \nexists e' . e \rightarrowtail^* e'$. By the definition of $\mathcal{E}$, if an expression belongs to $\mathcal{E}[\tau]$ for some τ, then it is strongly normalizing. The key step in the proof is to show that if an expression e types to τ, then $e \in \mathcal{E}[\tau]$.

For the complete example and a more detailed introduction to logical relations, we refer the reader to the *Introduction to Logical Relations* by Lau Skorstengaard [Sko19].

2.3. StreamCore

This section reviews prior work on formalizing the type system and semantics of RT-Lola as StreamCore [Koh+25]. In Chapter 3, we extend these definitions to support conditional streams, and consequently periodic streams.

Intuitively, a StreamCore specification consists of a set of stream equations defined over input and output *stream variables* from the sets $\mathbb{V}_{in}$ and $\mathbb{V}_{out}$. For simplicity, stream values are assumed to be integers, though the formalization can be extended to richer value domains. Semantically, each stream equation specifies a constraint on the value of a stream based on its past values or those of other streams. The semantics of StreamCore are defined in terms of *models* of a specification, where a model assigns a value to every stream at every time step such that all stream equations are satisfied.

2.3.1. Syntax

Formally, a specification is a list of stream equations of the form x `@` τ `:=` e, where x is an output stream variable in $\mathbb{V}_{out}$, τ is a pacing annotation, and e is a stream expression. A pacing annotation is a positive boolean formula over the input stream variables $\mathbb{V}_{in}$. Expressions may be constants $v \in \mathbb{Z}$, arithmetic operations such as addition, or stream accesses. Stream accesses take one of three forms:

- direct synchronous access, given by a target stream variable $x \in \mathbb{V}_{in} \uplus \mathbb{V}_{out}$,

- access to the previous value of a stream, written as x`.prev(`e`)`, where e is a fallback expression evaluated if x has no previous value, or
- asynchronous hold access, written as x`.hold(`e`)`, where e is a fallback expression evaluated if x has no value.

The complete syntax of StreamCore is provided in Def. 2.1.

Definition 2.1 (StreamCore Syntax)

Variables	$x \in \mathbb{V}_{in} \uplus \mathbb{V}_{out}$
Values	$v \in \mathbb{Z}$
Stream Expressions	$e ::= v \mid x \mid x\texttt{.prev(}e\texttt{)} \mid x\texttt{.hold(}e\texttt{)} \mid e_1 \texttt{ + } e_2 \mid \ldots$
Pacing Annotations	$\tau ::= x \in \mathbb{V}_{in} \mid \top \mid \tau_1 \wedge \tau_2 \mid \tau_1 \vee \tau_2$
Equations	$eq ::= x \texttt{ @ } \tau \texttt{ := } e$
Specifications	$S ::= \epsilon \mid eq \cdot S$

2.3.2. Semantics

In StreamCore, specifications are given a relational semantics, where models relate input streams to output streams. Before defining the relational semantics, we introduce auxiliary notions for streams and maps.

Definition 2.2 (Stream)

Streams are sequences of *optional* values, where the absence of a value is noted by $\bot$.

$$\mathit{Stream} \triangleq (\mathbb{Z} \uplus \{\bot\})^{\mathbb{N}}$$

For the formal definition of relational semantics in StreamCore, we require *StreamMaps*, which assign stream variables to streams.

Definition 2.3 (Stream Map)

Given a set of stream variables $X \subseteq \mathbb{V}_{in} \uplus \mathbb{V}_{out}$, the set of stream maps over X is defined as:

$$\mathit{Smap}(X) \triangleq \mathit{Stream}^X$$

Let $\rho \in \mathit{Smap}(X)$, the domain of ρ is denoted as $\mathit{dom}(\rho) \triangleq X$. Let $\emptyset$ be the unique map in $\mathit{Smap}(\emptyset)$. Given a stream variable x and $w \in \mathit{Stream}$, $(x \mapsto w) \in \mathit{Smap}(\{x\})$ denotes

the singleton map associating w to x. Finally, the *join* of two maps $\rho_X \in \mathit{Smap}(X)$ and $\rho_Y \in \mathit{Smap}(Y)$ as $\rho_X \cdot \rho_Y$ is defined as follows: $\rho_X \cdot \rho_Y \in \mathit{Smap}(X \cup Y)$ as follows:

$$\rho_1 \cdot \rho_2 \triangleq \lambda x. \begin{cases} \rho_1(x) & \text{if } x \in \mathit{dom}(\rho_1) \\ \rho_2(x) & \text{if } x \notin \mathit{dom}(\rho_1) \text{ and } x \in \mathit{dom}(\rho_2) \end{cases}$$

Note that $\cdot$ is not commutative in general, but only when X and Y are disjoint. In the following, ρ_{in} and ρ_{out} range over $\mathit{Smap}(\mathbb{V}_{in})$ and $\mathit{Smap}(\mathbb{V}_{out})$, respectively.

Semantics of Stream Expressions. Based on the previous definitions, the semantics of stream expressions are given as follows. For a stream map $\rho \in \mathit{Smap}(\mathbb{V}_{in} \uplus \mathbb{V}_{out})$ and a time point $n \in \mathbb{N}$, each expression e is assigned a value $[\![e]\!]^n_\rho \in \mathbb{Z} \uplus \{\lightning\}$. The symbol $\lightning$ denotes that the evaluation of the expression *failed*. To define the meaning of stream accesses, we introduce the following helper functions. Let $w \in \mathit{Stream}$ be a stream of optional values, $n \in \mathbb{N}$ a time point, and $v \in \mathbb{Z} \cup \lightning$.

Definition 2.4 (Stream Operators)

$$\mathit{Sync}(w, n) \triangleq \begin{cases} w(n) & \text{if } w(n) \neq \bot \\ \lightning & \text{if } w(n) = \bot \end{cases}$$

$$\mathit{Last}(w, n) \triangleq \begin{cases} w(n) & \text{if } w(n) \neq \bot \\ \mathit{Last}(w, n-1) & \text{if } w(n) = \bot \wedge n > 0 \\ \lightning & \text{if } w(n) = \bot \wedge n = 0 \end{cases}$$

$$\mathit{Hold}(w, n, v) \triangleq \begin{cases} \mathit{Last}(w, n) & \text{if } \mathit{Last}(w, n) \neq \lightning \\ v & \text{if } \mathit{Last}(w, n) = \lightning \end{cases}$$

$$\mathit{Prev}(w, n, v) \triangleq \begin{cases} \mathit{Last}(w, n-1) & \text{if } w(n) \neq \bot \wedge n > 0 \wedge \mathit{Last}(w, n-1) \neq \lightning \\ v & \text{if } w(n) \neq \bot \\ & \wedge (n = 0 \vee n > 0 \wedge \mathit{Last}(w, n-1) = \lightning) \\ \lightning & \text{if } w(n) = \bot \end{cases}$$

Intuitively, the functions can be described as follows:

- $\mathit{Sync}(w, n)$ returns the integer value of stream w at time n, or $\lightning$ if w is undefined at n.

- *Last*(w, n) returns the most recent defined value of w at or before time n. If no such value exists (i.e., w has never been different from $\bot$ up to and including n), then *Last*(w, n) fails and yields $\lightning$.
- *Hold*(w, n, v) also returns the most recent defined value of w at or before n. Unlike *Last*(w, n), if no such value exists (*Last*$(w, n) = \lightning$), it returns the default value v.
- *Prev*(w, n, v) corresponds to the `x.prev(e)` stream access. This operation first inspects the current value of stream x. If x has a defined value at time n, it returns the previous defined value of x. If no previous value exists (i.e., n is the first time x is defined), it returns the default value e. Formally, *Prev*(w, n, v) checks whether $w(n)$ is defined. If $w(n) = \bot$, then *Prev* fails and returns $\lightning$. If $w(n)$ is defined, *Prev* returns the last defined value of w excluding $w(n)$. If n is the first point where w is defined, *Prev* returns the default value v.

Definition 2.5 (Denotation of stream expressions)

Using these functions the semantics of stream expressions is defined as follows:

$$\begin{aligned}
[\![c]\!]^n_\rho &\triangleq c \\
[\![e_1 \texttt{ + } e_2]\!]^n_\rho &\triangleq [\![e_1]\!]^n_\rho +_\lightning [\![e_2]\!]^n_\rho \\
[\![x]\!]^n_\rho &\triangleq \mathit{Sync}(\rho(x), n) \\
[\![y\texttt{.prev(}e\texttt{)}]\!]^n_\rho &\triangleq \mathit{Prev}(\rho(y), n, [\![e]\!]^n_\rho) \\
[\![x\texttt{.hold(}e\texttt{)}]\!]^n_\rho &\triangleq \mathit{Hold}(\rho(y), n, [\![e]\!]^n_\rho)
\end{aligned}$$

Where the operator $+_\lightning$ is defined as follows:

$$v_1 +_\lightning v_2 \triangleq \begin{cases} v_1 + v_2 & \text{if } v_1 \neq \lightning \wedge v_2 \neq \lightning \\ \lightning & \text{otherwise} \end{cases}$$

Constant values and arithmetic operations are evaluated in the standard way, and, as expected, stream accesses are interpreted using the functions *Sync*, *Prev*, and *Hold*.

Semantics of Pacing Annotations. Before defining the semantics of complete specifications, we specify the semantics of pacing annotations. Intuitively, a pacing annotation represents a subset of discrete time points $x \subseteq \mathbb{N}$ at which a stream must take a defined value. Since pacing annotations are syntactically expressed as positive boolean formulas over input stream variables $\mathbb{V}_{in}$, their semantics depend on the given input stream map ρ_{in}. An atomic proposition in such a formula, corresponding to an input stream variable, denotes the set of time points where the associated stream in ρ_{in} has a defined value. The semantics of boolean connectives then follow naturally from the set operations $\cup$ and $\cap$.

Definition 2.6 (Denotation of pacing annotations)

Given an input stream map ρ_{in}, the semantics of pacing annotations are defined as follows:

$$\begin{aligned}
[\![x]\!]_{\rho_{in}} &\triangleq \{n \mid \rho_{in}(x)(n) \neq \bot\} \\
[\![\top]\!]_{\rho_{in}} &\triangleq \mathbb{N} \\
[\![\tau_1 \vee \tau_2]\!]_{\rho_{in}} &\triangleq [\![\tau_1]\!]_{\rho_{in}} \cup [\![\tau_2]\!]_{\rho_{in}} \\
[\![\tau_1 \wedge \tau_2]\!]_{\rho_{in}} &\triangleq [\![\tau_1]\!]_{\rho_{in}} \cap [\![\tau_2]\!]_{\rho_{in}}
\end{aligned}$$

Semantics of Specifications Finally, the semantics of specifications can be defined as a set of stream maps the satisfies the constraints imposed by the stream equations.

Definition 2.7 (Denotation of Equations)

$$\begin{aligned}
[\![x \texttt{ @ } \tau \texttt{ := } e]\!] &\triangleq \{\, \rho = \rho_{in} \cdot \rho_{out} \mid \forall n \in [\![\tau]\!]_{\rho_{in}}.\rho_{out}(x)(n) = [\![e]\!]^n_\rho \,\} \\
[\![\epsilon]\!] &\triangleq \mathit{Smap}(\mathbb{V}_{in}) \cdot \mathit{Smap}(\mathbb{V}_{out}) \\
[\![\mathit{eq} \cdot S]\!] &\triangleq [\![\mathit{eq}]\!] \cap [\![S]\!]
\end{aligned}$$

The denotation of a single equation is defined as the set of all stream maps in which the output stream variable of the equation equals the value of its associated stream expression. This requirement applies only at time points where the pacing annotation holds. Since $\rho_{out}(x)(n) \in \mathbb{Z} \cup \bot$ and $[\![e]\!] \in \mathbb{Z} \cup \lightning$, both x and e must evaluate to a well-defined integer value at all time points determined by the pacing annotation τ.

Safe Specifications. A definition for safe specifications is derived based on the semantics presented above.

Definition 2.8 (Safety)

A specification S is called *safe* if, for every set of input streams ρ_{in}, there exists at least one compatible set of output streams ρ_{out}. Formally,

$$\mathit{Safe}(S) \triangleq \forall \rho_{in} \in \mathit{Smap}(\mathbb{V}_{in}).\exists \rho_{out} \in \mathit{Smap}(\mathbb{V}_{out}).\ \rho_{in} \cdot \rho_{out} \in [\![S]\!]$$

This definition does not require uniqueness of the output stream map; thus, non-deterministic specifications are considered safe. Safety ensures that no stream access refers to a non-existing value. If such an access were possible, the evaluation would yield $\lightning$, which is not a valid stream value, and no compatible output stream map could exist in the denotational semantics.

2.3.3. Typing Rules of StreamCore

The typing rules of StreamCore operate at two levels: type judgments for specifications, denoted $\vdash$, and type judgments for expressions, denoted $\vdash^x$, where x records the stream variable associated with the expression.

Definition 2.9 (Typing Rules of StreamCore)

The type inference rules for specifications are defined as follows:

$$\textsc{Empty}\ \frac{}{\Gamma \vdash \epsilon} \qquad \textsc{Eq}\ \frac{x \notin dom(\Gamma) \qquad \Gamma \vdash^x e : \tau \qquad \Gamma, x : \tau \vdash \varphi}{\Gamma \vdash (x\ \texttt{@}\ \tau\ \texttt{:=}\ e) \cdot \varphi}$$

The type inference rules for expressions are defined as:

$$\textsc{Const}\ \frac{v \in \mathbb{Z}}{\Gamma \vdash^x v : \tau} \qquad \textsc{BinOp}\ \frac{\Gamma \vdash^x e_1 : \tau_{must} \qquad \Gamma \vdash^x e_2 : \tau_{must}}{\Gamma \vdash^x e_1\ \texttt{+}\ e_2 : \tau_{must}}$$

$$\textsc{DirectOut}\ \frac{y \in \mathbb{V}_{out} \qquad x \neq y \qquad (y : \tau_{can}) \in \Gamma \qquad \tau_{must} \models \tau_{can}}{\Gamma \vdash^x y : \tau_{must}} \qquad \textsc{DirectIn}\ \frac{y \in \mathbb{V}_{in} \qquad \tau_{must} \models y}{\Gamma \vdash^x y : \tau_{must}}$$

$$\textsc{HoldOut}\ \frac{x \neq y \qquad y \in dom(\Gamma) \qquad \Gamma \vdash^x e : \tau_{must}}{\Gamma \vdash^x y\texttt{.hold(}e\texttt{)} : \tau_{must}} \qquad \textsc{HoldIn}\ \frac{y \in \mathbb{V}_{in} \qquad \Gamma \vdash^x e : \tau_{must}}{\Gamma \vdash^x y\texttt{.hold(}e\texttt{)} : \tau_{must}}$$

$$\textsc{PrevOut}\ \frac{x \neq y \qquad (y : \tau_{can}) \in \Gamma \qquad \Gamma \vdash^x e : \tau_{must} \qquad \tau_{must} \models \tau_{can}}{\Gamma \vdash^x y\texttt{.prev(}e\texttt{)} : \tau_{must}}$$

$$\textsc{PrevIn}\ \frac{y \in \mathbb{V}_{in} \qquad \Gamma \vdash^x e : \tau_{must} \qquad \tau_{must} \models y}{\Gamma \vdash^x y\texttt{.prev(}e\texttt{)} : \tau_{must}} \qquad \textsc{Self}\ \frac{x = y \qquad \Gamma \vdash^x e : \tau_{must}}{\Gamma \vdash^x y\texttt{.prev(}e\texttt{)} : \tau_{must}}$$

Where $\tau_{must} \models \tau_{can}$ is defined as:

$$\tau_{must} \models \tau_{can} \triangleq \forall \rho_{in}.[\![\tau_{must}]\!]\rho_{in} \subseteq [\![\tau_{can}]\!]\rho_{in}$$

The inference rules for specifications process equations sequentially, adding the association between each stream variable and its pacing annotation to the typing environment. The Empty rule states that any typing context is valid for an empty specification. The Eq rule processes a single stream equation, recurses on the remaining specification, and binds the current stream variable x to its pacing annotation τ. To prevent multiple definitions of the same stream, it requires that x is not yet bound in the typing environment. Moreover, it ensures that the stream expression can be evaluated at pacing τ, expressed by the premise $\Gamma \vdash^{x} e : \tau$.

For stream expressions, the main type constraints arise from stream accesses. Each stream access can be typed using one of two rules, depending on whether the accessed stream is an input or an output stream.

Synchronous accesses, handled by the rules DirectOut, DirectIn, PrevOut, and PrevIn, require that the time points at which an expression *must* be evaluated are contained within the time points at which the accessed stream *can* be evaluated. This condition is expressed by $\tau_{must} \models \tau_{can}$. For output stream accesses, an additional restriction applies: the accessed stream must differ from the stream associated with the expression, thereby prohibiting self-access. The exception is the Self rule, which explicitly permits a stream to access its own past values through the `y.prev(e)` operator. This rule requires only that the default expression e can be evaluated at the same time points as the access itself, a condition shared with the PrevOut and PrevIn rules and the `y.hold(e)` operator.

The asynchronous accesses using `y.hold(e)` imposes a weaker requirement: the accessed stream y must simply be present in the domain of Γ and different from x, without any pacing constraint.

2.3.4. Logical Relations for Pacing Types

This section defines the semantic interpretation of pacing types using logical relations. Before introducing logical relations for pacing types, a relaxed semantics for stream expressions over partial stream maps is provided. This is necessary because type inference is an incremental process, where type definitions are gradually added to the typing environment. The logical relations connect pacing types and *typing environments* to stream semantics, including stream maps. Thus, it is essential to specify how stream expressions evaluate under partial stream maps in which some streams may be undefined.

In the partial semantics, a stream access operator explicitly returns $\lightning$ if the accessed stream is not in the domain of the map. The semantics also employ a single-value *memory cell*, denoted $x = v$, which records the last value of a stream. This primarily affects the semantics of the `y.prev(e)` operator when the accessed stream is the same as

the stream associated with the expression. The memory cell is later required to prove the type safety for streams that access their own past values. The full partial semantics is given below as follows:

Definition 2.10 (Semantics of Stream Expressions under Partial Maps)

$$\widehat{[\![c]\!]}_\rho^{n,x=v} \triangleq c$$

$$\widehat{[\![e_1 \texttt{ + } e_2]\!]}_\rho^{n,x=v} \triangleq \widehat{[\![e_1]\!]}_\rho^{n,x=v} +_\lightning \widehat{[\![e_2]\!]}_\rho^{n,x=v}$$

$$\widehat{[\![y]\!]}_\rho^{n,x=v} \triangleq \begin{cases} \mathit{Sync}(\rho(y), n) & \text{if } y \neq x \wedge y \in \mathit{dom}(\rho) \\ \lightning & \text{if } y = x \vee y \notin \mathit{dom}(\rho \end{cases}$$

$$\widehat{[\![y\texttt{.prev(}e\texttt{)}]\!]}_\rho^{n,x=v} \triangleq \begin{cases} \mathit{Prev}(\rho(y), n, \widehat{[\![e]\!]}_\rho^{n,x=v}) & \text{if } y \neq x \wedge y \in \mathit{dom}(\rho) \\ \lightning & \text{if } y \neq x \wedge y \notin \mathit{dom}(\rho) \\ v & \text{if } y = x \wedge n > 0 \wedge v \neq \lightning \\ \widehat{[\![e]\!]}_\rho^{n,x=v} & \text{if } y = x \wedge (n = 0 \vee v = \lightning) \end{cases}$$

$$\widehat{[\![x\texttt{.hold(}e\texttt{)}]\!]}_\rho^{n,x=v} \triangleq \begin{cases} \mathit{Hold}(\rho(y), n, \widehat{[\![e]\!]}_\rho^{n,x=v}) & \text{if } y \neq x \wedge y \in \mathit{dom}(\rho) \\ \lightning & \text{if } y = x \vee (y \neq x \wedge y \notin \mathit{dom}(\rho)) \end{cases}$$

The logical relations are defined with respect to the partial semantics of stream expressions. Since specification safety is formulated in terms of the total semantics, Lemma 5.2 in [Koh+25] shows that the partial semantics coincides with the total semantics for every extension of partial stream map to a total one. For completeness, we restate Lemma 5.2 of the paper:

Lemma 1. *Let* $x \in \mathbb{V}_{out}$, $Y \subseteq \mathbb{V}_{out} \setminus \{x\}$, *and* $\rho \in \mathit{Smap}(\mathbb{V}_{in} \cup Y)$ *and* $\rho' \in \mathit{Smap}(\mathbb{V}_{out} \setminus Y \setminus \{x\})$. *Then for any stream* $w \in \mathit{Stream}$ *and any* $n \in \mathbb{N}$, *we have:*

$$w(n) \neq \bot \implies \widehat{[\![e]\!]}_\rho^{n,x=\widetilde{\mathit{Last}}(w,n-1)} \neq \lightning \implies \widehat{[\![e]\!]}_\rho^{n,x=\widetilde{\mathit{Last}}(w,n-1)} = [\![e]\!]_{\rho \cdot (x \mapsto w) \cdot \rho'}$$

Where $\widetilde{\mathit{Last}}(w, n)$ *is defined as follows:*

$$\widetilde{\mathit{Last}}(w, n) \triangleq \begin{cases} \mathit{Last}(w, n) & \text{if } n \geqslant 0 \\ \lightning & \text{if } n < 0 \end{cases}$$

Logical Relations. In the following, four relations are introduced: $\mathcal{W}[\![-]\!]$, $\mathcal{G}[\![-]\!]$, $\mathcal{E}[\![-]\!]$, and $\mathcal{S}[\![-]\!]$ Here, $\mathcal{W}[\![-]\!]$ associates pacing annotations with streams, $\mathcal{G}[\![-]\!]$ associates typing contexts with output stream maps, $\mathcal{E}[\![-]\!]$ associates typing contexts with expressions, and $\mathcal{S}[\![-]\!]$ associates typing contexts with specifications.

Definition 2.11 (Interpretation of Pacing Types)

$$\mathcal{W}[\![\tau]\!]_{\rho_{\mathtt{in}}} \triangleq \{\, w \in \mathit{Stream} \mid \forall n \in [\![\tau]\!]^{n}_{\rho_{\mathtt{in}}}.w(n) \neq \bot \,\}$$

$$\mathcal{G}[\![\Gamma]\!]_{\rho_{in}} \triangleq \{\, \rho_{out} \in \mathit{Smap}(\mathit{dom}(\Gamma)) \mid \forall (x : \tau) \in \Gamma.\ \rho_{out}(x) \in \mathcal{W}[\![\tau]\!]_{\rho_{\mathtt{in}}} \,\}$$

$$\mathcal{E}[\![\Gamma \mid x : \tau]\!]_{\rho_{in}} \triangleq \{\, e \mid \forall \rho_{out} \in \mathcal{G}[\![\Gamma]\!]_{\rho_{in}}.\forall n \in [\![\tau]\!]_{\rho_{in}}.\forall v \in \mathbb{Z} \cup \{\lightning\}.\ \widehat{[\![e]\!]}^{n,x=v}_{\rho_{in}\cdot\rho_{out}} \neq \lightning \,\}$$

$$\mathcal{S}[\![\Gamma]\!]_{\rho_{\mathtt{in}}} \triangleq \{\, S \mid \forall \rho^{1}_{\mathtt{out}} \in \mathcal{G}[\![\Gamma]\!]_{\rho_{\mathtt{in}}}.\exists \rho^{2} \in \mathit{Smap}(\mathbb{V}_{out} \setminus \mathit{dom}(\Gamma)).\rho_{in} \cdot \rho^{1}_{\mathtt{out}} \cdot \rho^{2}_{\mathtt{out}} \in [\![S]\!] \,\}$$

The stream relation $\mathcal{W}[\![\tau]\!]_{\rho_{in}}$ s the simplest of the four relations. It contains all streams w that have a defined value at least at the time points specified by $[\![\tau]\!]_{\rho_{in}}$. The relation $\mathcal{G}[\![\Gamma]\!]_{\rho_{in}}$ extends this notion to typing contexts: it contains all partial maps $\rho_{out} \in \mathit{Smap}(\mathit{dom}(\Gamma))$ such that, for every $x : \tau$ in Γ, the stream $\rho_{out}(x)$ satisfies the interpretation of τ. The relation $\mathcal{E}[\![\Gamma \mid x : \tau]\!]_{\rho_{in}}$ contains all expressions e that can be evaluated at pacing τ for any output stream x consistent with Γ. Last but not least, the specification relation $\mathcal{S}[\![\Gamma]\!]_{\rho_{in}}$ contains all specifications S for which any partial stream map in $\mathcal{G}[\![\Gamma]\!]_{\rho_{in}}$ can be extended to a total stream map that is a solution of S.

Semantic Typing. As outlined in Sect. 2.2, semantic typing judgments are defined on the basis of these logical relations.

$$\Gamma \models S \triangleq \forall \rho_{in}.\ S \in \mathcal{S}[\![\Gamma]\!]_{\rho_{in}}$$

$$\Gamma \models \rho \triangleq \forall \rho_{in}.\ \rho \in \mathcal{G}[\![\Gamma]\!]_{\rho_{in}}$$

$$\Gamma \models^{x} e : \tau \triangleq \forall \rho_{in}.e \in \mathcal{E}[\![\Gamma \mid x : \tau]\!]_{\rho_{in}}$$

The proof of type system soundness, i.e., $\emptyset \vdash S \implies \mathit{Safe}(S)$, proceeds in two steps. First, it is shown that semantically well-typed specifications are safe ($\emptyset \models S \implies \mathit{Safe}(S)$). Second, it is established that syntactic well-typedness implies semantic well-typedness, i.e., $\emptyset \vdash S \implies \emptyset \models S$. For the complete proofs, we refer the reader to the original work on pacing type soundness [Koh+25].

Chapter 3

Proving Type-Safety for Conditional Streams

In this chapter, we extend the existing StreamCore formalization to establish type safety for conditional streams. The syntax of StreamCore is first modified to include conditional stream equations, and the value domain of streams is extended with boolean values to capture the semantics of conditions. Semantically, a conditional equation restricts the time points at which a stream must produce a defined value to those where both its pacing holds and the condition evaluates to `true`.

As illustrated in Sect. 1.1 with Spec. 1.7, conditional streams introduce further risks for safe monitoring. To mitigate these risks, we extend the StreamCore type system to permit conditional output streams in pacing annotations. Within the pacing type system, conditional streams are hence treated similar to input streams and determine their own pacing. The type system enforces that any stream synchronously accessing a conditional stream x must be annotated with at least the pacing x, ensuring evaluation occurs only when x produces a value.

3.1. StreamCore with Conditional Streams

The extension of StreamCore proceeds in three steps. First, the syntax is enriched with conditional equations and references to conditional output streams in pacing annotations. Second, the semantics of pacing annotations are adapted to account for these references, and semantics for conditional streams are introduced. Third, additional type inference rules are defined to handle conditional output streams.

3.1.1. Syntax

Formally, the syntax of StreamCore is extended in three ways compared to the definition presented in Def. 2.1:

1. The value domain of streams is extended to include booleans, $\mathbb{B} \triangleq \texttt{true}, \texttt{false}$.
2. Conditional stream equations are added.
3. Most importantly, pacing annotations are extended to allow references to conditional output stream variables.

To distinguish between output streams with evaluation conditions and those without, we define the set of conditional output stream variables as $\mathbb{V}_{cond} \subseteq \mathbb{V}_{out}$. If necessary, the set of non-conditional output stream variables is denoted by $\mathbb{V}_{\texttt{true}}$. The complete syntax of StreamCore with conditional streams is provided in Def. 3.1.

Definition 3.1 (StreamCore Syntax with Conditional Streams)

Variables	$x \in \mathbb{V}_{in} \uplus \mathbb{V}_{out}$	
Values	$v \in \mathbb{Z} \cup \mathbb{B}$	
Stream Expressions	$e ::= v \mid x \mid x\texttt{.prev(e)} \mid x\texttt{.hold(e)} \mid e_1 \texttt{ + } e_2 \mid \dots$	
Pacing Annotations	$\tau ::= x \in \mathbb{V}_{in} \uplus \mathbb{V}_{cond} \mid \top \mid \tau_1 \wedge \tau_2 \mid \tau_1 \vee \tau_2$	
Equations	$eq ::= x \texttt{ @ } \tau \texttt{ := } e$	for $x \in \mathbb{V}_{\texttt{true}}$
Conditional Equations	$eq_c ::= x \texttt{ @ } \tau \texttt{ when } e_c \texttt{ := } e$	for $x \in \mathbb{V}_{cond}$
Specifications	$S ::= \epsilon \mid eq \cdot S$	

Values are defined as the union of integers and booleans. For simplicity, we assume throughout this thesis that all specifications are well-typed with respect to data types, i.e., arithmetic operations are applied only to integers and boolean connectives only to booleans.

Atomic propositions in pacing annotations are extended to range over both input streams and conditional output streams. A *conditional equation* includes an additional expression e_c, introduced by the `when` keyword following the pacing annotation. This expression specifies the condition under which the stream must produce a defined value and is assumed to evaluate to a boolean.

3.1.2. Semantics

We start by adapting Def. 2.2 such that streams are defined over integers and boolean values:

Definition 3.2 (Stream)

Streams are sequences of *optional* values, where the absence of a value is noted by $\bot$.

$$\mathit{Stream} \triangleq (\mathbb{Z} \uplus \mathbb{B} \uplus \{\bot\})^{\mathbb{N}}$$

Next the updated semantics of pacing annotations is presented that accounts for conditional output streams:

Definition 3.3 (Denotation of Pacing Annotations)

Given a stream map ρ, the semantics of pacing annotations are defined as follows:

$$\begin{aligned}
[\![x]\!]_\rho &\triangleq \{n \mid \rho(x)(n) \neq \bot\} \\
[\![\top]\!]_\rho &\triangleq \mathbb{N} \\
[\![\tau_1 \vee \tau_2]\!]_\rho &\triangleq [\![\tau_1]\!]_\rho \cup [\![\tau_2]\!]_\rho \\
[\![\tau_1 \wedge \tau_2]\!]_\rho &\triangleq [\![\tau_1]\!]_\rho \cap [\![\tau_2]\!]_\rho
\end{aligned}$$

The updated semantics of pacing annotations differ from Def. 2.6 only in that the stream map is now total over both input and output streams. Since the syntax and semantics of expressions remain unchanged, we adapt the semantics of specifications in Def. 2.7 to incorporate conditional streams.

Definition 3.4 (Denotation of Equations)

$$\begin{aligned}
[\![x \texttt{ @ } \tau \texttt{ := } e]\!] &\triangleq \{\ \rho = \rho_{in} \cdot \rho_{out} \mid \forall n \in [\![\tau]\!]_\rho.\ \rho_{out}(x)(n) = [\![e]\!]^n_\rho\ \} \\
[\![x \texttt{ @ } \tau \texttt{ when } e_c \texttt{ := } e]\!] &\triangleq \{\ \rho = \rho_{in} \cdot \rho_{out} \mid \forall n \in [\![\tau]\!]_\rho.\ [\![e_c]\!]^n_\rho \neq \lightning \\
&\qquad\qquad \wedge ([\![e_c]\!]^n_\rho = \texttt{true} \implies \rho_{out}(x)(n) = [\![e]\!]^n_\rho)\ \} \\
[\![\epsilon]\!] &\triangleq \mathit{Smap}(\mathbb{V}_{in}) \cdot \mathit{Smap}(\mathbb{V}_{out}) \\
[\![eq \cdot S]\!] &\triangleq [\![eq]\!] \cap [\![S]\!] \\
[\![eq_c \cdot S]\!] &\triangleq [\![eq_c]\!] \cap [\![S]\!]
\end{aligned}$$

The denotational semantics of a conditional equation x `@` τ `when` e_c `:=` e coincide with those of unconditional streams, except for the additional premise that requires the condition to evaluate to a defined value, reflecting the intuition that evaluation is only possible when both the expression and the condition can be evaluated. If the condition evaluates to `true`, the stream must equal the value defined by its expression; otherwise, the model may assign any value, including $\bot$.

Note that this requirement is in fact a relaxation of the unconditional semantics, as the time points at which a stream must be equal to its expression are further constrained.

3.1.3. Typing Rules for StreamCore with Conditional Streams

To formalize type safety, we introduce a new typing relation for pacing annotations. An annotation is well-typed if all stream variables it references are present in the typing environment Γ, ensuring that only well-typed streams occur in pacing annotations. We then extend the type inference rules to handle boolean values and add a dedicated rule for conditional streams.

Typing Pacing Annotations. Formally, a pacing annotation is well-typed with respect to Γ if it can be derived using the following inference rules:

$$\textsc{Var}\ \frac{x \in dom(\Gamma)}{\Gamma \vdash x} \qquad \textsc{Top}\ \frac{}{\Gamma \vdash \top}$$

$$\textsc{And}\ \frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \wedge \tau_2} \qquad \textsc{Or}\ \frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \vee \tau_2}$$

The inference rules for conjunction and disjunction recurse on the operands in the expected manner. The central case is the Var rule, which requires that any stream variable occurring in a pacing annotation is contained in the environment.

The type inference rules for expressions remain largely unchanged compared to Def. 2.9. The main difference is that the Const rule now also allows boolean values. Boolean operations are typed using the BinOp rule.

The type inference rules for stream expressions are then given as follows:

$$\textsc{Const}\ \frac{v \in \mathbb{Z} \cup \mathbb{B}}{\Gamma \vdash^{x} v : \tau} \qquad \textsc{BinOp}\ \frac{\Gamma \vdash^{x} e_1 : \tau_{must} \qquad \Gamma \vdash^{x} e_2 : \tau_{must}}{\Gamma \vdash^{x} e_1 \texttt{ + } e_2 : \tau_{must}}$$

$$\textsc{DirectOut}\ \frac{y \in \mathbb{V}_{out} \qquad x \neq y \qquad (y : \tau_{can}) \in \Gamma \qquad \tau_{must} \models \tau_{can}}{\Gamma \vdash^{x} y : \tau_{must}} \qquad \textsc{DirectIn}\ \frac{y \in \mathbb{V}_{in} \qquad \tau_{must} \models y}{\Gamma \vdash^{x} y : \tau_{must}}$$

$$\textsc{HoldOut}\ \frac{x \neq y \qquad y \in dom(\Gamma) \qquad \Gamma \vdash^{x} e : \tau_{must}}{\Gamma \vdash^{x} y\texttt{.hold(}e\texttt{)} : \tau_{must}} \qquad \textsc{HoldIn}\ \frac{y \in \mathbb{V}_{in} \qquad \Gamma \vdash^{x} e : \tau_{must}}{\Gamma \vdash^{x} y\texttt{.hold(}e\texttt{)} : \tau_{must}}$$

$$\textsc{PrevOut}\ \frac{x \neq y \qquad (y : \tau_{can}) \in \Gamma \qquad \Gamma \vdash^{x} e : \tau_{must} \qquad \tau_{must} \models \tau_{can}}{\Gamma \vdash^{x} y\texttt{.prev(}e\texttt{)} : \tau_{must}}$$

$$\textsc{PrevIn}\ \frac{y \in \mathbb{V}_{in} \qquad \Gamma \vdash^{x} e : \tau_{must} \qquad \tau_{must} \models y}{\Gamma \vdash^{x} y\texttt{.prev(}e\texttt{)} : \tau_{must}} \qquad \textsc{Self}\ \frac{x = y \qquad \Gamma \vdash^{x} e : \tau_{must}}{\Gamma \vdash^{x} y\texttt{.prev(}e\texttt{)} : \tau_{must}}$$

We adapt the definition of $\tau_{must} \models \tau_{can}$ to account for output streams:

$$\tau_{must} \models \tau_{can} \triangleq \forall \rho. [\![\tau_{must}]\!]_\rho \subseteq [\![\tau_{can}]\!]_\rho$$

Type Inference for Specifications. On the specification level, we add a typing rule for conditional streams EqC.

$$\textsc{Empty}\ \frac{}{\Gamma \vdash \epsilon} \qquad \textsc{Eq}\ \frac{x \notin dom(\Gamma) \qquad \Gamma \vdash \tau \qquad \Gamma \vdash^{x} e : \tau \qquad \Gamma, x : \tau \vdash \varphi}{\Gamma \vdash (x\texttt{ @ }\tau\texttt{ := }e) \cdot \varphi}$$

$$\textsc{EqC}\ \frac{x \notin dom(\Gamma) \qquad \Gamma \vdash \tau \qquad \Gamma \vdash^{x} e : \tau \qquad \Gamma \vdash^{x} e_c : \tau \qquad \Gamma, x : x \vdash \varphi}{\Gamma \vdash x\texttt{ @ }\tau\texttt{ when }e_c\texttt{ := }e \cdot \varphi}$$

At the specification level, we add the EqC rule for conditional streams. Unlike the Eq rule, EqC additionally requires that the evaluation condition e_c is compatible with the pacing τ. Moreover, instead of associating the stream variable with its declared pacing type, EqC associates it with its own stream variable. Consequently, any stream y that accesses a conditional stream x synchronously must include at least pacing x in its annotation. This observation is crucial for establishing type safety in the next section.

Both the EQ and EQC rules require that pacing annotations are well-typed, ensuring that no annotation refers to an output stream that is not yet typed.

Self Accesses in Conditions. Before turning to the interpretation of pacing types through logical relations, we highlight an unintuitive but safe edge case: a stream that accesses its own past value in its condition. Such cases may yield unexpected models in the semantics. Consider the following specification:

```
1 output x @true when x.prev(false) := false
```

The issue arises because the value of `x.prev(false)` at time n is defined only if x itself is defined at n, as specified in Def. 2.5. If the condition evaluates to a defined value at n, it must be `false`. Thus, the condition of the stream is `false`, and the stream is not required to produce a value. In this situation, `x.prev(false)` is not guaranteed to evaluate to a defined value, contradicting the initial assumption. However, due to the semantics of conditional streams, a valid model exists in which the stream evaluates to the value of its expression whenever the condition is `false`. As a result, the constant-`false` stream is a valid interpretation of x in this example. Although unexpected from a user perspective, this behavior does not compromise type safety.

3.2. Proving Type Safety of Conditional Streams

To prove type safety of StreamCore with conditional streams, we first define the denotational semantics of pacing types over partial maps. We then adapt the logical relations from Def. 2.11 to this setting and use them to prove type safety. An overview of the proof is provided in Fig. 3.1.

We begin with the interpretation of pacing annotations over partial maps.

Definition 3.5 (Denotation of Pacing Annotations over Partial Maps)

$$\widehat{[\![\tau]\!]}_\rho \triangleq [\![\tau]\!]_{\rho \cdot (\lambda\_.\lambda\_.\bot)}$$

These semantics follow Def. 3.3, except that any stream not in the domain of the partial map is assigned to the constant $\bot$ stream, yielding a total map. This allows us to derive the following lemma for pacing types:

Lemma 2. *Let ρ be a stream map such that: $\rho \in Smap(\mathbb{V}_{in}) \cdot Smap(dom(\Gamma))$ for a typing context Γ. Then it holds:*

$$\forall x \in (\mathbb{V}_{out} \setminus dom(\Gamma)).\forall w \in Stream.\Gamma \vdash \tau \implies \widehat{[\![\tau]\!]}_\rho = \widehat{[\![\tau]\!]}_{\rho \cdot (x \mapsto w)}$$

Proof. Proof by induction over the structure of pacing types. We only show the interesting base case when $\tau = y$ is a stream variable. Suppose $\Gamma \vdash y$ for a stream variable y. It remains to show that:

$$\widehat{[\![y]\!]}_\rho = \widehat{[\![y]\!]}_{\rho \cdot (x \mapsto w)}$$

From $\Gamma \vdash y$ follows that $y \in dom(\rho)$. Suppose that $\rho(y) = w_y$ where w_y is an arbitrary stream. From the definition of the $\cdot$ operator on stream maps given in Def. 2.3 and $y \in dom(\rho)$ it follows that: $(\rho \cdot (x \mapsto w))(y) = w_y$. It immediately follows that:

$$\widehat{[\![y]\!]}_\rho = \{\, n \mid w_y(n) \neq \bot \,\} = \{\, n \mid (\rho \cdot (x \mapsto w))(y)(n) \neq \bot \,\} = \widehat{[\![y]\!]}_{\rho \cdot (x \mapsto w)}$$

□

The lemma states that under the partial semantics for pacing annotations if an annotation is well-typed it does not matter how the partial map is extended, the set of time points for which the pacing annotation holds remains the same.

To establish type safety, we must also determine when the partial semantics of pacing annotations agree with their denotational semantics. Intuitively, they agree for a stream map ρ and its totalization by $(\lambda.\lambda.\bot)$ whenever τ contains no stream variables outside the domain of ρ. This property is enforced by the typing relation for pacing annotations introduced above. The following lemma formalizes this agreement between partial and denotational semantics:

Lemma 3. *Let Γ be a typing context and $\rho \in Smap(\mathbb{V}_{in}) \cdot Smap(dom(\Gamma))$ be a partial stream map. Then it holds for any pacing annotation τ:*

$$\forall \rho_{out}{}' \in Smap(\mathbb{V}_{out} \setminus dom(\Gamma)).\Gamma \vdash \tau \implies \widehat{[\![\tau]\!]}_\rho = [\![\tau]\!]_{\rho \cdot \rho_{out}{}'}$$

Proof. By induction on the type derivation tree of $\Gamma \vdash \tau$. □

Intuitively, the lemma holds because $\Gamma \vdash \tau$ guarantees that all variables in τ are in the domain of ρ. Thus, totalizing ρ does not affect the evaluation of τ, including the specific totalization used by the partial semantics.

3.2.1. Interpretation of Pacing Types over Input and Output Streams

Finally, we adapt the logical relations of Def. 2.11 to account for output streams in pacing annotations. As before, we define four logical relations that connect pacing types and typing contexts to semantic objects such as streams and stream maps. The four relations are defined as follows:

Definition 3.6 (Interpretation of Pacing Types)

$$\mathcal{W}[\![\tau]\!]_{\rho_{in}\cdot\rho_{out}} \triangleq \{\, w \in \mathit{Stream} \mid \forall n \in \widehat{[\![\tau]\!]}_{\rho_{in}\cdot\rho_{out}}.w(n) \neq \bot \,\}$$

$$\mathcal{G}[\![\Gamma]\!]_{\rho_{in}} \triangleq \{\, \rho_{out} \in \mathit{Smap}(\mathit{dom}(\Gamma)) \mid \forall (x:\tau) \in \Gamma.\ \rho_{out}(x) \in \mathcal{W}[\![\tau]\!]_{\rho_{in}\cdot\rho_{out}} \,\}$$

$$\begin{aligned}\mathcal{E}[\![\Gamma \mid x:\tau]\!]_{\rho_{in}} \triangleq \{\, e \mid\ & \forall \rho_{out} \in \mathcal{G}[\![\Gamma]\!]_{\rho_{in}}.\\ & \forall n \in \widehat{[\![\tau]\!]}_{\rho_{in}\cdot\rho_{out}}.\\ & \forall v \in \mathbb{Z} \cup \mathbb{B} \cup \{\lightning\}.\ \widehat{[\![e]\!]}^{n,x=v}_{\rho_{in}\cdot\rho_{out}} \neq \lightning \,\}\end{aligned}$$

$$\begin{aligned}\mathcal{S}[\![\Gamma]\!]_{\rho_{in}} \triangleq \{\, S \mid\ & \forall {\rho_{out}}^1 \in \mathcal{G}[\![\Gamma]\!]_{\rho_{in}}.\\ & \exists {\rho_{out}}^2 \in \mathit{Smap}(\mathbb{V}_{out} \setminus \mathit{dom}(\Gamma)).\rho_{in}\cdot{\rho_{out}}^1\cdot{\rho_{out}}^2 \in [\![S]\!] \,\}\end{aligned}$$

The first logical relation, $\mathcal{W}[\![\tau]\!]$, associates a pacing annotation τ with the set of streams that produce values according to this pacing. The main extension compared to the definition in [Koh+25] is that pacing annotations are now interpreted over both input and *output* streams. Accordingly, the relation is indexed not only by an input stream map ρ_{in} but also by an output stream map ρ_{out}. Since these maps may not include all output streams of the specification, i.e., they are partial output stream maps, the relation is defined using the partial semantics of pacing annotations introduced in Def. 3.5.

The second relation, $\mathcal{G}[\![\Gamma]\!]$, interprets typing contexts as the set of all partial output stream maps with the same domain as the context, such that every stream in the map is well-typed with respect to its annotation.

The third relation, $\mathcal{E}[\![\Gamma \mid x:\tau]\!]$, captures the set of stream expressions that can be safely evaluated in context Γ for a stream x with pacing τ. As pacing annotations are interpreted over partial output stream maps, the relation is defined with respect to their partial semantics. It is also based on the partial semantics of expressions from Def. 2.10, which remain unchanged by the introduction of conditional streams. Type safety results are then lifted from the partial semantics to the denotational semantics of expressions by Lem. 1.

The fourth relation, $\mathcal{S}[\![\Gamma]\!]$, defines the set of specifications that are safe to evaluate under typing context Γ, using the denotational semantics of specifications from Def. 3.4.

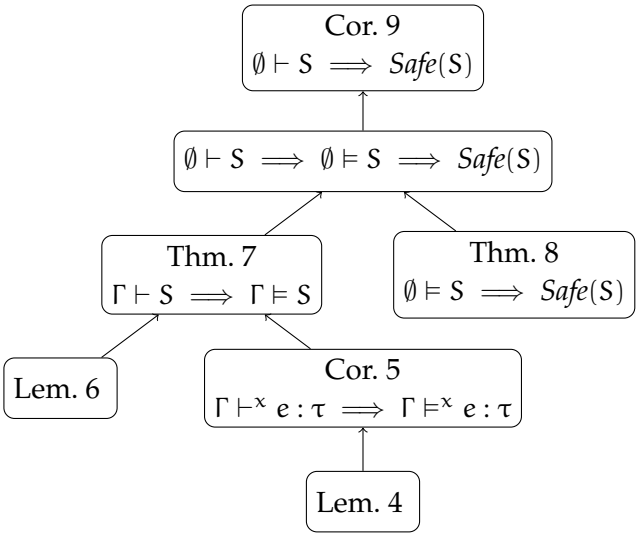


Figure 3.1.: A visualization of the Proof Structure.

Because these semantics operate only on total maps, the relation quantifies over all possible totalizations of the partial output stream map.

The definitions of semantic typing introduced in Sect. 2.3.4 remain unchanged.

3.2.2. Proof Overview

The overall proof follows the structure of [Koh+25], although the changes to the logical relations lead to different proofs. Fig. 3.1 illustrates how lemmas, corollaries, and theorems combine to establish type safety. The main result is stated in Cor. 9, which forms the root of the proof tree in the figure. This result is decomposed using logical relations and semantic typing: Thm. 7 shows that syntactically well-typed specifications are also semantically well-typed, and Thm. 8 establishes that semantically well-typed specifications are safe. The proof of Thm. 7 is divided into two steps. First, Cor. 5 shows that if an expression is syntactically well-typed in environment Γ, then it is also semantically well-typed under Γ, based on a case analysis of the typing rules for expressions in Lem. 4. Second, using this result and Lem. 6, which addresses the typing rules for specifications, we conclude that syntactic typing implies semantic typing.

3.2.3. Proofs in Detail

This section presents the detailed proof of type safety for StreamCore with conditional streams. The proof proceeds in the order shown in Fig. 3.1: beginning with the soundness of expression typing, then the soundness of specification typing, and finally combining these results to prove type safety as a whole.

Soundness of Expressions

The following lemma establishes the soundness of the type inference rules for expressions by showing that syntactic typing entails semantic typing.

Lemma 4 (Compatibility of Typing Rules for Expressions)**.** *The following implications hold:*

1. $v \in \mathbb{Z} \cup \mathbb{B} \implies \Gamma \vDash^{x} v : \tau_{must}$
2. $y \in \mathbb{V}_{in} \implies \tau_{must} \models y \implies \Gamma \vDash^{x} y : \tau_{must}$
3. $y \in \mathbb{V}_{out} \implies (y : \tau_{can}) \in \Gamma \implies \tau_{must} \models \tau_{can} \implies \Gamma \vDash^{x} y : \tau_{must}$
4. $\Gamma \vDash^{x} e_1 : \tau_{must} \implies \Gamma \vDash^{x} e_2 : \tau_{must} \implies \Gamma \vDash^{x} e_1 + e_2 : \tau_{must}$
5. $y \in \mathbb{V}_{in} \implies \Gamma \vDash^{x} e : \tau_{must} \implies \tau_{must} \models y \implies \Gamma \vDash^{x} y.\mathit{prev}(e) : \tau_{must}$
6. $\Gamma \vDash^{x} e : \tau_{must} \implies \Gamma \vDash^{x} x.\mathit{prev}(e) : \tau_{must}$
7. $\Gamma \vDash^{x} e : \tau_{must} \implies (y : \tau_{can}) \in \Gamma \implies \tau_{must} \models \tau_{can} \implies \Gamma \vDash^{x} y.\mathit{prev}(e) : \tau_{must}$
8. $y \in \mathbb{V}_{in} \implies \Gamma \vDash^{x} e : \tau_{must} \implies \Gamma \vDash^{x} y.\mathit{hold}(e) : \tau_{must}$
9. $y \in \mathit{dom}(\Gamma) \implies \Gamma \vDash^{x} (e : \tau_{must}) \implies \Gamma \vDash^{x} y.\mathit{hold}(e) : \tau_{must}$

Proof. To prove the above lemma, recall that $\Gamma \vDash^{x} e : \tau_{must}$ is defined as $\forall \rho_{in}.e \in \mathcal{E}[\![\Gamma \mid x : \tau_{must}]\!]$. We assume an arbitrary instantiation of the quantifiers in $\mathcal{E}[\![\Gamma \mid x : \tau_{must}]\!]$ as follows: Let $\rho_{out} \in \mathcal{G}[\![\Gamma]\!]$, let $n \in \widehat{[\![\tau_{must}]\!]}_{\rho_{in} \cdot \rho_{out}}$. Finally, the proofs of the individual cases boil down to showing that $\widehat{[\![e]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}} \neq \bot$, i.e. that the expression evaluates to a defined value given some partial evaluation model.

(1) Constants. Suppose $v \in \mathbb{Z} \uplus \mathbb{B}$ and let ρ_{in} be an arbitrary input stream map s.t. $\rho_{in} \in \mathit{Smap}(\mathbb{V}_{in})$. Pick any $p \in \mathbb{Z} \cup \mathbb{B} \cup \lightning$, then it remains to show that:

$$\widehat{[\![v]\!]}^{n,x=p}_{\rho_{in} \cdot \rho_{out}} \neq \lightning$$

By definition of $\widehat{[\![-]\!]}$ it holds that $\widehat{[\![v]\!]}^{n,x=p}_{\rho_{in} \cdot \rho_{out}} = v \neq \lightning$ as $v \in \mathbb{Z} \cup \mathbb{B}$

(2) Direct Input Stream Access. Suppose $y \in \mathbb{V}_{in}$ is an input stream variable and let ρ_{in} be an arbitrary input stream map s.t. $\rho_{in} \in \mathit{Smap}(\mathbb{V}_{in})$. Also suppose that $\tau_{must} \models y$. Pick any $p \in \mathbb{Z} \cup \mathbb{B} \cup \lightning$, then it remains to show that:

$$\widehat{[\![y]\!]}^{n,x=p}_{\rho_{in} \cdot \rho_{out}} \neq \lightning$$

As input streams have no defining equations, it holds that $x \neq y$ and therefore:

$$\widehat{[\![y]\!]}^{n,x=p}_{\rho_{in} \cdot \rho_{out}} = \mathit{Sync}((\rho_{in} \cdot \rho_{out})(y), n)$$

Because $\tau_{must} \models y$ it holds that $\forall \rho. [\![\tau_{must}]\!]_\rho \subseteq [\![y]\!]_\rho$. Therefore, it especially holds that:

$$[\![\tau_{must}]\!]_{\rho_{in} \cdot \rho_{out} \cdot \lambda\_.\lambda\_.\bot} \subseteq [\![y]\!]_{\rho_{in} \cdot \rho_{out} \cdot \lambda\_.\lambda\_.\bot} \equiv \widehat{[\![\tau_{must}]\!]}_{\rho_{in} \cdot \rho_{out}} \subseteq \widehat{[\![y]\!]}_{\rho_{in} \cdot \rho_{out}} \tag{3.1}$$

Because $\rho_{in} \in \mathit{Smap}(\mathbb{V}_{in})$ and $y \in \mathbb{V}_{in}$ it holds that $y \in \mathit{dom}(\rho_{in})$, and it follows that $\forall t \in \widehat{[\![y]\!]}_{\rho_{in} \cdot \rho_{out}}.(\rho_{in} \cdot \rho_{out})(y)(t) \neq \bot$. With that it especially holds that $(\rho_{in} \cdot \rho_{out})(y)(n) \neq \bot$. It follows that $\mathit{Sync}((\rho_{in} \cdot \rho_{out})(y), n) = (\rho_{in} \cdot \rho_{out})(y)(n) \neq \lightning$.

(3) Direct Output Stream Access. Let ρ_{in} be an arbitrary input stream map. Suppose $y \in \mathbb{V}_{out}$ is an output stream variable different from x, that $(y : \tau_{can}) \in \Gamma$ and that $\tau_{must} \models \tau_{can}$. Pick any $v \in \mathbb{Z} \cup \mathbb{B} \cup \lightning$. We must show that:

$$\widehat{[\![y]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}} \neq \lightning$$

First note that $y \in \mathit{dom}(\rho_{out})$ as $\rho_{out} \in \mathcal{G}[\![\Gamma]\!]$. Because $\rho_{out} \in \mathcal{G}[\![\Gamma]\!]$ and $(y : \tau_{can}) \in \Gamma$ it holds by definition of $\mathcal{G}[\![-]\!]$ that $\rho_{out}(y) \in \mathcal{W}[\![\tau_{can}]\!]_{\rho_{in} \cdot \rho_{out}}$. From the definition of $\mathcal{W}[\![\tau_{can}]\!]_{\rho_{in} \cdot \rho_{out}}$ it follows that:

$$\forall t \in \widehat{[\![\tau_{can}]\!]}_{\rho_{in} \cdot \rho_{out}}.(\rho_{in} \cdot \rho_{out})(y)(t) \neq \bot$$

Similar to the observation 3.1 it follows from $\tau_{must} \models \tau_{can}$ that

$$(\rho_{in} \cdot \rho_{out})(y)(n) \neq \bot$$

By definition of $\widehat{[\![-]\!]}$ and $y \neq x$ it follows that $\widehat{[\![y]\!]}^{n,x=p}_{\rho_{in} \cdot \rho_{out}} = \mathit{Sync}((\rho_{in} \cdot \rho_{out})(y), n) = (\rho_{in} \cdot \rho_{out})(y)(n) \neq \lightning$.

(4) Binary Operation. Let ρ_{in} be an arbitrary input stream map. Suppose $\Gamma \vDash^x e_1 : \tau_{must}$ and $\Gamma \vDash^x e_1 : \tau_{must}$. Let $v \in \mathbb{Z} \cup \mathbb{B} \cup \lightning$ be arbitrary. It follows that $\widehat{[\![e_1]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}} \neq \lightning$ and $\widehat{[\![e_2]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}} \neq \lightning$. It remains to show that:

$$\widehat{[\![e_1 + e_2]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}} \neq \lightning$$

By definition of $\widehat{[\![-]\!]}$ it follows: $\widehat{[\![e_1 + e_2]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}} = \widehat{[\![e_1]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}} +_\lightning \widehat{[\![e_2]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}} \neq \lightning$

(5) Past Input Stream Access. Let ρ_{in} be an arbitrary input stream map. Let $y \in \mathbb{V}_{in}$ be an input stream variable and therefore $y \neq x$. Suppose $\Gamma \vDash^x e : \tau_{must}$ and $\tau_{must} \models y$. Pick any $v \in \mathbb{Z} \cup \mathbb{B} \cup \lightning$. It remains to show that:

$$\widehat{[\![y\texttt{.prev(}e\texttt{)}]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}} \neq \lightning$$

Because $y \in \mathbb{V}_{in}$ it follows that $y \in dom(\rho_{in})$ and therefore $y \in dom(\rho_{in} \cdot \rho_{out})$. From the definition of $\widehat{[\![-]\!]}$ it follows that $\widehat{[\![y\texttt{.prev(}e\texttt{)}]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}} = Prev((\rho_{in} \cdot \rho_{out})(y), n, \widehat{[\![e]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}})$. Observe that $Prev(w, n, v)$ can only return $\lightning$ if $v = \lightning$ or $w(n) = \bot$. From $\Gamma \vDash^x e : \tau_{must}$ it follows that $v = \widehat{[\![e]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}} \neq \lightning$. From $\tau_{must} \models y$ it follows that $(\rho_{in} \cdot \rho_{out})(y)(n) \neq \bot$. It follows that:

$$\widehat{[\![y\texttt{.prev(}e\texttt{)}]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}} = Prev((\rho_{in} \cdot \rho_{out})(y), n, \widehat{[\![e]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}}) \neq \lightning$$

(6) Self Access. Let ρ_{in} be an arbitrary input stream map. Suppose $\Gamma \vDash^x e : \tau_{must}$. Pick any $v \in \mathbb{Z} \cup \mathbb{B} \cup \lightning$. We have to show that:

$$\widehat{[\![x\texttt{.prev(}e\texttt{)}]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}} \neq \lightning$$

By definition of $\widehat{[\![x\texttt{.prev(}e\texttt{)}]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}}$ is either equal to v if $v \neq \lightning$ which concludes the proof, or equal to $\widehat{[\![e]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}}$. Because $\Gamma \vDash^x e : \tau_{must}$ it holds that $\widehat{[\![e]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}} \neq \lightning$ and therefore it follows that $\widehat{[\![x\texttt{.prev(}e\texttt{)}]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}} \neq \lightning$.

(7) Past Output Stream Access. Let ρ_{in} be an arbitrary input stream map. Let $y \in \mathbb{V}_{out}$ be an output stream variable different from x. Suppose $\Gamma \vDash^x e : \tau_{must}$, $(y : \tau_{can}) \in \Gamma$ and $\tau_{must} \models \tau_{can}$. Pick any $v \in \mathbb{Z} \cup \mathbb{B} \cup \lightning$. It remains to show that:

$$\widehat{[\![y\texttt{.prev(}e\texttt{)}]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}} \neq \lightning$$

Because $(y : \tau_{can}) \in \Gamma$ it holds that $y \in dom(\rho_{out})$. Because $y \neq x$ it follows by definition of $\widehat{[\![-]\!]}$ that $\widehat{[\![y\texttt{.prev(}e\texttt{)}]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}} = Prev((\rho_{in} \cdot \rho_{out})(y), n, \widehat{[\![e]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}})$. As observed before $Prev(w, n, v)$ can only return $\lightning$ if $v = \lightning$ or $w(n) = \bot$. From $\Gamma \vDash^x e : \tau_{must}$ it follows that $v = \widehat{[\![e]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}} \neq \lightning$. Because $\tau_{must} \models \tau_{can}$ it holds that $n \in \widehat{[\![\tau_{can}]\!]}_{\rho_{in} \cdot \rho_{out}}$. Since $y \in dom(\rho_{out})$ it holds that $\rho_{out}(y)(n) \neq \bot$ and therefore $w(n) = (\rho_{in} \cdot \rho_{out})(y)(n) \neq \bot$. From this it follows that $\widehat{[\![y\texttt{.prev(}e\texttt{)}]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}} = Prev((\rho_{in} \cdot \rho_{out})(y), n, \widehat{[\![e]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}}) \neq \lightning$.

(8) Input Stream Hold. Let ρ_{in} be an arbitrary input stream map and let $v \in \mathbb{Z} \cup \mathbb{B} \cup \lightning$. Let $y \in \mathbb{V}_{in}$ be an input stream variable and therefore different from x. Suppose $\Gamma \vDash^x e : \tau_{must}$. We have to show that:

$$\widehat{[\![y\texttt{.hold(}e\texttt{)}]\!]}^{n,x=v}_{\rho_{in} \cdot \rho_{out}} \neq \lightning$$

As $y \neq x$ and $y \in dom(\rho_{in})$ it holds by definition that:

$$[\![y\,\widehat{\texttt{.hold(}e\texttt{)}}]\!]^{n,x=v}_{\rho_{in}\cdot\rho_{out}} = Hold((\rho_{in}\cdot\rho_{out})(y), n, \widehat{[\![e]\!]}^{n,x=v}_{\rho_{in}\cdot\rho_{out}})$$

Observe that $Hold(w, n, v)$ can only evaluate to $\lightning$ if $v = \lightning$ by definition. As $v = \widehat{[\![e]\!]}^{n,x=v}_{\rho_{in}\cdot\rho_{out}}$ and $\Gamma \vDash^x e : \tau_{must}$ it holds that: $v = \widehat{[\![e]\!]}^{n,x=v}_{\rho_{in}\cdot\rho_{out}} \neq \lightning$.

(9) Output Stream Hold. Let ρ_{in} be an arbitrary input stream map and let $v \in \mathbb{Z} \cup \mathbb{B} \cup \lightning$. Let $y \in \mathbb{V}_{out}$ be an output stream variable different from x. Suppose that $y \in dom(\Gamma)$ and $\Gamma \vDash^x e : \tau_{must}$. It remains to show that:

$$[\![y\,\widehat{\texttt{.hold(}e\texttt{)}}]\!]^{n,x=v}_{\rho_{in}\cdot\rho_{out}} \neq \lightning$$

Because $y \in dom(\Gamma)$ and therefore $y \in dom(\rho_{out})$ and $y \neq x$ it holds that:

$$[\![y\,\widehat{\texttt{.hold(}e\texttt{)}}]\!]^{n,x=v}_{\rho_{in}\cdot\rho_{out}} = Hold((\rho_{in}\cdot\rho_{out})(y), n, \widehat{[\![e]\!]}^{n,x=v}_{\rho_{in}\cdot\rho_{out}})$$

As before, because $\Gamma \vDash^x e : \tau_{must}$ it holds that $\widehat{[\![e]\!]}^{n,x=v}_{\rho_{in}\cdot\rho_{out}} \neq \lightning$ and therefore it follows that $[\![y\,\widehat{\texttt{.hold(}e\texttt{)}}]\!]^{n,x=v}_{\rho_{in}\cdot\rho_{out}} \neq \lightning$.

□

Using this lemma the proof of the overall expression soundness stated in the following corollary follows through induction over the type derivation tree:

Corollary 5 (Expression Soundness)**.**

$$\Gamma \vdash^x e : \tau \implies \Gamma \vDash^x e : \tau$$

Proof. By induction on the type derivation tree of $\Gamma \vdash^x e : \tau$ applying Lem. 4 for each case. □

Soundness of Specifications

Similarly to proof of soundness for expressions, Lem. 6 establishes the soundness of type inference rules for specifications by showing that they are consistent with semantic typing. It distinguishes a separate case for each rule.

Lemma 6 (Compatibility of Typing Rules for Specifications)**.** *The following statements hold true:*

1. $\Gamma \vDash \epsilon$
2. $x \notin dom(\Gamma) \implies \Gamma \vdash \tau \implies \Gamma, x : \tau \vDash S \implies \Gamma \vDash^x e : \tau$
   $\implies \Gamma \vDash (x\ \texttt{@}\ \tau\ \texttt{:=}\ e) \cdot S$

3. $x \notin dom(\Gamma) \implies \Gamma \vdash \tau \implies \Gamma, x : x \vDash S \implies \Gamma \vDash^x e_c : \tau \implies \Gamma \vDash^x e : \tau$
$\implies \Gamma \vDash (x \texttt{ @ } \tau \texttt{ when } e_c \texttt{ := } e) \cdot S$

Proof. Recap that $\Gamma \vDash S$ is defined as $\forall \rho_{in} \in Smap(\mathbb{V}_{in}).S \in \mathcal{S}[\![\Gamma]\!]_{\rho_{in}}$. Therefore we first fix an arbitrary $\rho_{in} \in Smap(\mathbb{V}_{in})$. To show that $S \in \mathcal{S}[\![\Gamma]\!]_{\rho_{in}}$ we have to show that prove that for any ${\rho_{out}}^1 \in \mathcal{G}[\![\Gamma]\!]_{\rho_{in}}$ there exists a ${\rho_{out}}^2 \in Smap(\mathbb{V}_{out} \setminus (\mathrm{dom})(\Gamma))$ such that $\rho_{in} \cdot {\rho_{out}}^1 \cdot {\rho_{out}}^2 \in [\![S]\!]$. Said otherwise: there is a totalization of the partial output stream map ${\rho_{out}}^1$ such that it satisfies the stream equations of the specification.

(1) Empty Specification. Let ${\rho_{out}}^1 \in \mathcal{G}[\![\Gamma]\!]_{\rho_{in}}$ be fixed but arbitrary. Note that by definition of $[\![-]\!]$ it holds that $[\![\epsilon]\!] = Smap(\mathbb{V}_{in}) \cdot Smap(\mathbb{V}_{out})$, i.e. for an empty specification any stream map is valid. Hence it suffices to pick any ${\rho_{out}}^2 \in Smap(\mathbb{V}_{out} \setminus dom(\Gamma))$ such as $\lambda\_.\lambda\_.\bot.$, i.e. the stream map the maps every stream to a stream of constant $\bot$.

(2) Unconditional Stream. Suppose $x \notin dom(\Gamma)$,$\Gamma \vdash \tau$, $\Gamma, x : \tau \vDash S$ and $\Gamma \vDash^x e : \tau$. Let ${\rho_{out}}^1 \in \mathcal{G}[\![\Gamma]\!]_{\rho_{in}}$ be fixed but arbitrary. It remains to show the existence of an output stream map ${\rho_{out}}^2 \in Smap(\mathbb{V}_{out} \setminus dom(\Gamma))$ such that:

$$\rho_{in} \cdot {\rho_{out}}^1 \cdot {\rho_{out}}^2 \in [\![(x \texttt{ @ } \tau \texttt{ := } e) \cdot S]\!] = [\![x \texttt{ @ } \tau \texttt{ := } e]\!] \cap [\![S]\!]$$

From $\Gamma \vDash^x e : \tau$ it follows that:

$$\forall n \in \widehat{[\![\tau]\!]}_{\rho_{in} \cdot {\rho_{out}}^1}.\forall v \in \mathbb{Z} \cup \mathbb{B} \cup \{\lightning\}.\widehat{[\![e]\!]}^{n, x=v}_{\rho_{in} \cdot {\rho_{out}}^1} \neq \lightning \tag{3.2}$$

Using this, we define a stream w_e for e recursively and show that it is in $\mathcal{W}[\![\tau]\!]_{\rho_{in} \cdot {\rho_{out}}^1}$.

$$w_e(n) \triangleq \begin{cases} \widehat{[\![e]\!]}^{n, x=\widetilde{Last}(w_e(0) \cdot \ldots \cdot w_e(n-1) \cdot 0^\omega, n-1)}_{\rho_{in} \cdot {\rho_{out}}^1} & \text{if } n \in \widehat{[\![\tau]\!]}_{\rho_{in} \cdot {\rho_{out}}^1} \\ \bot & \text{if } n \notin \widehat{[\![\tau]\!]}_{\rho_{in} \cdot {\rho_{out}}^1} \end{cases}$$

First, observe that w_e is well-defined, because $w_e(n)$ only depends on w_e at time points $n' < n$. Because of observation 3.2 it holds that $\forall n \in \widehat{[\![\tau]\!]}_{\rho_{in} \cdot {\rho_{out}}^1}.w_e(n) \neq \lightning$. It follows that $w_e \in \mathcal{W}[\![\tau]\!]_{\rho_{in} \cdot {\rho_{out}}^1}$. Using Lem. 2, $x \notin dom(\Gamma)$ and $\Gamma \vdash \tau$ it follows that $w_e \in \mathcal{W}[\![\tau]\!]_{\rho_{in} \cdot {\rho_{out}}^1 \cdot (x \mapsto w_e)}$. By definition of $\mathcal{G}[\![-]\!]$ it follows that ${\rho_{out}}^1 \cdot (x \mapsto w_e) \in \mathcal{G}[\![\Gamma, x : \tau]\!]_{\rho_{in}}$. Observe that by definition $Last(w, n)$ only depends on positions n' of stream w where $n' \leqslant n$. Therefore, it holds that:

$$\widetilde{Last}(w_e(0) \cdot \ldots \cdot w_e(n-1) \cdot 0^\omega, n-1) = \widetilde{Last}(w_e, n-1)$$

Using this and applying Lem. 1 we derive that:

$$\forall n \in \widehat{[\![\tau]\!]}_{\rho_{in} \cdot {\rho_{out}}^1}.w_e(n) = \widehat{[\![e]\!]}^{n, x=\widetilde{Last}(w_e, n-1)}_{\rho_{in} \cdot {\rho_{out}}^1} = [\![e]\!]^n_{\rho_{in} \cdot {\rho_{out}}^1 \cdot (x \mapsto w_e) \cdot {\rho_{out}}'} \tag{3.3}$$

It remains to show that there is a $\rho_{out}{}^2$ such that $\rho_{in} \cdot \rho_{out}{}^1 \cdot \rho_{out}{}^2 \in [\![x \texttt{ @ } \tau \texttt{ := } e]\!]$ and $\rho_{in} \cdot \rho_{out}{}^1 \cdot \rho_{out}{}^2 \in [\![S]\!]$. Using $\Gamma, x : \tau \vDash S$ and picking $\rho_{out}{}^1 \cdot (x \mapsto w_e)$ for the first all quantifier, we obtain a $\rho_{out}{}^S \in \mathit{Smap}(\mathbb{V}_{out} \setminus \mathit{dom}(\Gamma, x : \tau))$ such that $\rho_{in} \cdot \rho_{out}{}^1 \cdot (x \mapsto w_e) \cdot \rho_{out}{}^S \in [\![S]\!]$. We pick $(x \mapsto w_e) \cdot \rho_{out}{}^S$ as $\rho_{out}{}^2$ and it remains to show that $\rho_{in} \cdot \rho_{out}{}^1 \cdot \rho_{out}{}^2 \in [\![x \texttt{ @ } \tau \texttt{ := } e]\!]$. Based on observation 3.3 and applying Lem. 3 using $\Gamma \vdash \tau$ we derive that:

$$\forall n \in [\![\tau]\!]_{\rho_{in} \cdot \rho_{out}{}^1 \cdot \rho_{out}{}^2}.w_e(n) = [\![e]\!]^n_{\rho_{in} \cdot \rho_{out}{}^1 \cdot (x \mapsto w) \cdot \rho_{out}{}'}$$

Because $x \notin \mathit{dom}(\rho_{out}{}^S)$ it holds that $(\rho_{in} \cdot \rho_{out}{}^1 \cdot \rho_{out}{}^2)(x) = w_e$ and it follows that $\rho_{in} \cdot \rho_{out}{}^1 \cdot \rho_{out}{}^2 \in [\![x \texttt{ @ } \tau \texttt{ := } e]\!]$.

(3) Conditional Stream. Suppose $x \notin \mathit{dom}(\Gamma)$,$\Gamma \vdash \tau$, $\Gamma, x : x \vDash S$, $\Gamma \vDash^x e_c : \tau$ and $\Gamma \vDash^x e : \tau$. Let $\rho_{out}{}^1 \in \mathcal{G}[\![\Gamma]\!]_{\rho_{in}}$ be fixed but arbitrary. It remains to show the existence of an output stream map $\rho_{out}{}^2 \in \mathit{Smap}(\mathbb{V}_{out} \setminus \mathit{dom}(\Gamma))$ such that:

$$\rho_{in} \cdot \rho_{out}{}^1 \cdot \rho_{out}{}^2 \in [\![(x \texttt{ @ } \tau \texttt{ when } e_c \texttt{ := } e) \cdot S]\!] = [\![x \texttt{ @ } \tau \texttt{ when } e_c \texttt{ := } e]\!] \cap [\![S]\!]$$

We construct a stream $w_e \in \mathcal{W}[\![\tau]\!]_{\rho_{in} \cdot \rho_{out}{}^1 \cdot (x \mapsto w_e)}$ in the exact same way as for an unconditional stream. However, it it not obvious, whether it holds that:

$$\rho_{out}{}^1 \cdot (x \mapsto w_e) \in \mathcal{G}[\![\Gamma, x : x]\!]_{\rho_{in}}$$

Because $\rho_{out}{}^1 \in \mathcal{G}[\![\Gamma]\!]_{\rho_{in}}$ it holds that $\rho_{out}{}^1 \cdot (x \mapsto w_e) \in \mathit{dom}(\Gamma, x : x)$. It remains to show that $\forall (x' : \tau') \in (\Gamma, x : x).(\rho_{out}{}^1 \cdot (x \mapsto w_e))(x') \in \mathcal{W}[\![\tau]\!]_{\rho_{in} \cdot \rho_{out}{}^1 \cdot (x \mapsto w_e)}$. As $\rho_{out}{}^1 \in \mathcal{G}[\![\Gamma]\!]_{\rho_{in}}$ this holds for all $x' \neq x \in \Gamma, x : x$ and therefore it suffices to show that:

$$(\rho_{out}{}^1 \cdot (x \mapsto w_e))(x) \in \mathcal{W}[\![x]\!]_{\rho_{in} \cdot \rho_{out}{}^1 \cdot (x \mapsto w_e)}$$

By definition of $\mathcal{W}[\![-]\!]$ it must hold that $\forall n \in \widehat{[\![x]\!]}_{\rho_{in} \cdot \rho_{out}{}^1 \cdot (x \mapsto w_e)}.w_e(n) \neq \bot$. Because $x \notin \mathit{dom}(\rho_{in} \cdot \rho_{out}{}^1)$ it holds bz definition of $\widehat{[\![-]\!]}$ that:

$$\widehat{[\![x]\!]}_{\rho_{in} \cdot \rho_{out}{}^1 \cdot (x \mapsto w_e)} = \{\, n \mid w_e(n) \neq \bot \,\}$$

It follows immediately, that $\forall n \in \{\, n \mid w_e(n) \neq \bot \,\}.w_e(n) \neq \bot$ and therefore $w_e \in \mathcal{W}[\![x]\!]_{\rho_{in} \cdot \rho_{out}{}^1 \cdot (x \mapsto w_e)}$. Consequently, it holds that:

$$\rho_{out}{}^1 \cdot (x \mapsto w_e) \in \mathcal{G}[\![\Gamma, x : x]\!]_{\rho_{in}}$$

Similar to before, we use $\Gamma, x : x \vDash S$ and pick $\rho_{out}{}^1 \cdot (x \mapsto w_e)$ for the first quantifier in $\mathcal{S}[\![\Gamma, x : x]\!]$ to obtain an $\rho_{out}{}^S \in \mathit{Smap}(\mathbb{V}_{out} \setminus \mathit{dom}(\Gamma, x : x))$ such that $\rho_{in} \cdot \rho_{out}{}^1 \cdot (x \mapsto w_e) \cdot \rho_{out}{}^S \in [\![S]\!]$. We pick $(x \mapsto w_e) \cdot \rho_{out}{}^S$ as $\rho_{out}{}^2$ and it remains to show that:

$$\rho_{in} \cdot \rho_{out}{}^1 \cdot \rho_{out}{}^2 \in [\![x \texttt{ @ } \tau \texttt{ when } e_c \texttt{ := } e]\!]$$

Because $\Gamma \vDash^x e_c : \tau$ and using Lem. 1 we can derive that:

$$\forall n \in \widehat{[\![\tau]\!]}_{\rho_{in} \cdot \rho_{out}{}^1} . \widehat{[\![e_c]\!]}^{n, x = \widetilde{Last}(w_e, n-1)}_{\rho_{in} \cdot \rho_{out}{}^1} = [\![e_c]\!]^n_{\rho_{in} \cdot \rho_{out}{}^1 \cdot \rho_{out}{}^2} \neq \lightning$$

Using Lem. 3 and $\Gamma \vdash^x \tau$ we can show that"

$$\forall n \in [\![\tau]\!]_{\rho_{in} \cdot \rho_{out}{}^1 \cdot \rho_{out}{}^2} . [\![e_c]\!]^n_{\rho_{in} \cdot \rho_{out}{}^1 \cdot \rho_{out}{}^2} \neq \lightning \tag{3.4}$$

Because of observation 3.3 from before it also holds that:

$$\forall n \in [\![\tau]\!]_{\rho_{in} \cdot \rho_{out}{}^1 \cdot \rho_{out}{}^2} . w_e(n) = [\![e]\!]^n_{\rho_{in} \cdot \rho_{out}{}^1 \cdot \rho_{out}{}^2} \tag{3.5}$$

Note that this holds independently of whether e_c evaluates to `true` or `false` at time n. Therefore it follows from observation 3.4 and 3.5 that:

$$\rho_{in} \cdot \rho_{out}{}^1 \cdot \rho_{out}{}^2 \in [\![x \texttt{ @ } \tau \texttt{ when } e_c \texttt{ := } e]\!]$$

□

Using this Lem. 6 the soundness of specifications follows by induction over the type derivation tree of a specification, applying Cor. 5 to the premisses of the inference rules.

Theorem 7 (Specification Soundness)**.**

$$\Gamma \vdash S \implies \Gamma \vDash S$$

Proof. By induction on the type derivation tree of $\Gamma \vdash S$.
Base Case: Let $S = \epsilon$ and suppose $\Gamma \vdash S$. By case (1) of Lem. 6 it immediately follows that $\Gamma \vDash S$.
Inductive Cases:

1. Let $S' = (x \texttt{ @ } \tau \texttt{ := } e) \cdot S$ and suppose that $\Gamma \vdash S'$, i.e. $x \notin dom(\Gamma)$, $\Gamma \vdash^x e : \tau$ and that $\Gamma, x : \tau \vdash S$. By induction, it holds that $\Gamma, x : \tau \vDash S$. From Cor. 5 it follows that $\Gamma \vDash^x e : \tau$. Applying case (2) of Lem. 6 it follows that $\Gamma \vDash S'$.

2. Let $S' = (x \texttt{ @ } \tau \texttt{ when } e_c \texttt{ := } e) \cdot S$ and suppose that $\Gamma \vdash S'$, i.e. $x \notin dom(\Gamma)$, $\Gamma \vdash^x e : \tau$, $\Gamma \vdash^x e_c : \tau$ and that $\Gamma, x : x \vdash S$. By induction, it holds that $\Gamma, x : x \vDash S$. From Cor. 5 it follows that $\Gamma \vDash^x e : \tau$ and that $\Gamma \vDash^x e_c : \tau$. Applying case (3) of Lem. 6 it follows that $\Gamma \vDash S'$.

□

3.2.4. Type Safety

Before type safety can be proven, it must be shown that it follows from the soundness of the type inference rules established above. Thm. 8 establishes this and states that, if a specification is semantically well typed in the empty environment, then it implies that there exists and output stream map for every possible input stream map, i.e. the specification is safe.

Theorem 8 (Safety)**.**

$$\emptyset \vDash S \implies \mathit{Safe}(S)$$

Proof. Let $\emptyset \vDash S$ and let $\rho_{in} \in \mathit{Smap}(\mathbb{V}_{in})$. By the definition $\mathit{Safe}(S)$ it remains to show that there exists a $\rho_{out} \in \mathit{Smap}(\mathbb{V}_{out})$ such that $\rho_{in} \cdot \rho_{out} \in [\![S]\!]$. By definition of $\emptyset \vDash S$ it holds that $S \in \mathcal{S}[\![\emptyset]\!]_{\rho_{in}}$. By definition of $\mathcal{S}[\![\emptyset]\!]_{\rho_{in}}$ it holds that:

$$\forall {\rho_{out}}^1 \in \mathcal{G}[\![\emptyset]\!]_{\rho_{in}}.\exists {\rho_{out}}^2 \in \mathit{Smap}(\mathbb{V}_{out}).\rho_{in} \cdot {\rho_{out}}^1 \cdot {\rho_{out}}^2 \in [\![S]\!]$$

By fixing ${\rho_{out}}^1 = \emptyset$, it holds that $\rho_{in} \cdot {\rho_{out}}^1 \cdot {\rho_{out}}^2 = \rho_{in} \cdot \emptyset \cdot {\rho_{out}}^2 = \rho_{in} \cdot {\rho_{out}}^2 \in [\![S]\!]$. We conclude the proof by picking ${\rho_{out}}^2$ for ρ_{out}. □

Finally, Thm. 8 and Thm. 7 can be combined to derive the type safety result for conditional streams in StreamCore.

Corollary 9 (Type Safety)**.**

$$\emptyset \vdash S \implies \mathit{Safe}(S)$$

Proof. Suppose $\emptyset \vdash S$. By Thm. 7 it follows that $\emptyset \vDash S$. Applying Thm. 8 $\mathit{Safe}(S)$ follows. □

Chapter 4

Encoding Real-Time Streams

In this chapter, we present two constructions that translate the real-time features of RTLola into the extended StreamCore framework with conditional streams. First, we define how periodically paced streams can be encoded in StreamCore. Second, we show how sliding window aggregations can be represented using a sequence of periodic streams that compute the buckets of the sliding window, as discussed in Sect. 2.1.2.

4.1. Periodic Streams

To encode periodic streams in StreamCore, we assume the existence of a clocked input stream that evaluates to a defined value at a fixed frequency. All periodic streams depend on this clock input. Their *when* conditions restrict the evaluation points of the clock input to match the desired frequency.

Formally, let $\mathbb{P}$ be the set of periodic streams in the RTLola specification, with frequencies $f_1, \ldots, f_k \in \mathbb{N}$ (in Hertz) and expressions $e_1, \ldots, e_k$. Let f' be the frequency of the clock input stream, defined as $lcm(f_1, \ldots, f_k)$, where *lcm* denotes the least common multiple. We denote this clock input stream by *clock*:

```
1 // Assumed to produce values at the frequency f'
2 input clock: Bool
```

We assume that all added stream names are fresh, i.e., they do not occur in the original RTLola specification. To further restrict the evaluation points of periodic streams, we introduce the following auxiliary output stream:

```
1 output step @clock := (step.prev(0) + 1) % f'
```

This stream, `step`, continuously counts from 0 to $f' - 1$ using the modulo operator `%` whenever the `clock` input evaluates. It is used to further subdivide the frequency of the `clock` into the required frequencies for the periodic output streams. For each $f_i \in f_1, \ldots, f_k$, we construct an output stream `clock_`f_i that evaluates at frequency f_i.

```
output clock_f_i @clock when step % (f' / f_i) == 0 := true
```

Intuitively, this holds true as:

$$f' \,\%\, (f'/f_i) = 0 \Leftrightarrow c * (f'/f_i) + 0 = f' \Leftrightarrow c = f_i$$

This construction ensures that $\texttt{step}\%(f'/f_i) = 0$ is satisfied exactly f_i times as `step` cycles from 0 to $f' - 1$.

However, naively translating of a periodic output stream $s \in \mathbb{P}$ with frequency f_i and expression e as follows would prevent specifications from being type safe when periodic streams reference one another.

```
output s @clock_f_i := e
```

Since periodic streams are encoded as conditional streams, every referenced stream must appear in the pacing annotation of the referencing stream. Intuitively, references between periodic streams should be allowed if the referencing stream runs at a lower frequency than the referenced one. To ensure type safety, we extend the pacing annotation of each stream with the clock streams of all synchronously accessed periodic dependencies:

Definition 4.1 (Encoding Periodic Streams)

Formally, let $s \in \mathbb{P}$ evaluate at frequency f_s with expression e. Let $\mathbb{X} \subseteq \mathbb{P}$ be the set of periodic streams accessed synchronously in e, and let $\mathbb{C}$ be the set of clock streams that form their pacing annotations. Then s is encoded in StreamCore as follows:

```
output s @clock_f_i ∧ ⋀_{c∈ℂ} c := e
```

This construction guarantees type safety by ensuring that each periodic stream evaluates at no more than its assigned frequency. However, the encoded stream is not guaranteed to evaluate exactly at its frequency. Instead, due to the inclusion of synchronous dependencies in the pacing annotation, the evaluation occurs at the greatest common divisor (gcd) of its own frequency and those of its synchronous dependencies. To enforce exact frequencies, one would need to restrict the set $\mathbb{X}$ to contain only those periodic streams f^* such that $gcd(f_i, f^*) = f_i$.

As an example, consider a specification with streams annotated at disjoint frequencies.

```
output a @2Hz := true
output b @3Hz := true
```

Following the above construction, we introduce a `clock` input stream that evaluates at $lcm(2,3) = 6$ Hertz. The `step` stream then counts from 0 to 5 at each event of `clock`, resetting to 0 every full second. The resulting encoding of the specification is as follows:

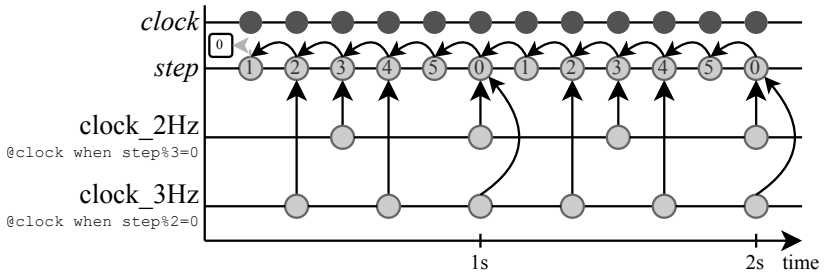


Figure 4.1.: An Example Trace Visualization of the Specification in Spec. 4.1.

```
1  // Assumed to produce values at the frequency 6Hz
2  input clock: Bool
3
4  output step @clock := (step.prev(0) + 1) % 6
5
6  output clock_2Hz @clock when step % 3 == 0 := true
7  output clock_3Hz @clock when step % 2 == 0 := true
8
9  output a @clock_2Hz := true
10 output b @clock_3Hz := true
```

Specification 4.1: The Encoding of Streams with Different Frequencies.

A visualization of `clock_2Hz` and `clock_3Hz` is given in Fig. 4.1. The figure illustrates how conditional evaluation based on the `step` stream produces the desired sub-frequencies from the assumed frequency of the `clock` input.

4.2. Sliding Windows

In this section, we present an encoding of sliding windows in StreamCore. We implement the bucket-based algorithm described in Sect. 2.1.2, which evaluates sliding windows using periodic streams.

To this end, we introduce a new stream access, `x.fresh()`. It returns whether stream `x` produced a value in the current time step. In other words, it is an asynchronous access that evaluates to `true` if the current value exists and to `false` otherwise. Formally, its semantics are defined as follows:

$$[\![\texttt{x.fresh()}]\!]^n_\rho \triangleq \begin{cases} \texttt{true} & \text{if } \rho(x)(n) \neq \bot \\ \texttt{false} & \text{if } \rho(x)(n) = \bot \end{cases}$$

The typing rules for `x.fresh()` are identical to those of the `hold` operator, except that no default expression is required. Consequently, its soundness proof proceeds analogously to that of `hold`.

Using this operator, we encode sliding windows as follows. Suppose a specification contains a stream x that aggregates values of stream y:

```
output x @f_xHz := y.aggregate(over: ds, using: agr)
```

We assume that the clock streams defined in Sect. 4.1 are available and that $d \geqslant \frac{1}{f}$, i.e., the sliding window spans at least one period. We denote the neutral element of the aggregation function `agr` by ϵ.

The sliding window is expanded as follows.

Definition 4.2 (Encoding Sliding Windows)

Let b be the number of buckets, computed as $b = d * f_x$. For $1 \leqslant i \leqslant b-1$, let the stream representing bucket b_i be defined as:

```
output b_i @clock_f_x := b_{i+1}.prev(ϵ)
```

Let the bucket b_b be defined as follows:

```
output b_b @clock_f_x ∨ y :=
   if clock_f_x.fresh() then
      if y.fresh() then
         y.hold(ϵ)
      else
         ϵ
   else
     agr(b_b.prev(ϵ), y.hold(ϵ))
```

Finally, the buckets are aggregated to construct the value of the window:

```
output x @clock_f_x := agr(b_1.prev(ϵ), ..., b_b.prev(ϵ))
```

The last bucket, b_b, aggregates the values of y. The remaining buckets $b_1, \ldots, b_{b-1}$ propagate this aggregated value by accessing the previous value of the next bucket at each clock cycle. The aggregation of y uses a disjunctive pacing annotation that evaluates either at each clock cycle or when y produces a value. The `fresh` operator distinguishes between these two cases.

When evaluated at a clock cycle, the bucket reinitializes for the next period: it takes either the current value of y or ϵ if no value exists. By checking whether a value of y exists using `y.fresh()`, the first window bound is inclusive in time, i.e., the aggregation window at time t covers all time points i with $t - d \leqslant i < t$.

When the stream does not evaluate at a clock cycle (the `else` branch), it must be the case that y produced a value, which is then aggregated with the previous bucket value.

Thus, if the clock evaluates at real-time point t, bucket b_b contains the value of y at time t, or ϵ if none exists. Bucket b_{b-1} contains all aggregated values of y for times i with $t - (1/f_x) \leqslant i < t$. Bucket b_{b-2} aggregates all values for $t - 2(1/f_x) \leqslant i < t - (1/f_x)$, and so on.

All generated streams, except b_b, share the annotation @clock_f_x and only access each other. Bucket b_b includes the additional disjunct $\vee y$. Its type safety follows because all stream accesses within it are asynchronous, except the `.prev(·)` self-access, which is safe under the SELF inference rule. The synchronous accesses to buckets in b_{b-1} and x are also valid, since clock_$f_x \models$ clock_$f_x \vee y$.

4.3. An Example

We illustrate this translation using Spec. 1.4, which detects sensor failures by counting received values over a fixed period via a sliding window aggregation.

```
1  input position: Float
2  input rpm: Float
3  // Assumed to produce values at 1Hz
4  input clock: Bool
5
6  output step @clock := (step.prev(0) + 1) % 1
7
8  // Constructed clock streams
9  output clock_1Hz @clock when step % 1 == 0 := true
10
11 output velocity @position := (position - position.prev(position)) * 36.0
12 output trigger_too_fast @position when velocity > 150 := "Driving too fast"
13
14 // Sliding window buckets
15 output b_60 @clock_1Hz ∨ rpm :=
16       if clock_1Hz.fresh() then
17          if rpm.fresh() then
18             1
19          else
20             0
21       else
22          b_60.prev(0) + 1
23 output b_59 @clock_1Hz := b_60.prev(0)
24        ⋮
25 output b_2 @clock_1Hz := b_3.prev(0)
26 output b_1 @clock_1Hz := b_2.prev(0)
27
28 output count @clock_1Hz:= b_1.prev(0) + ... + b_60.prev(0)
29 output trigger_rpm_failure @clock_1Hz when count < 60 := "RPM Sensor Failure"
30
31 output shift @position ∨ rpm := velocity.hold(0) < 30 && rpm.hold(or: 0) > 3000
32 output trigger_shift @position ∨ rpm when shift := "Shift to higher gear"
```

Specification 4.2: An Encoding of Spec. 1.4 into STREAMCORE with Conditional Streams

Because the specification uses a single frequency (1Hz) for periodic pacing, the generated clock_1Hz is semantically identical to the clock input stream, assumed to tick at 1Hz.

Trigger conditions are encoded as conditional streams. For instance, the trigger in line 12 of the specification encodes its condition `velocity > 150` directly as the evaluation condition. A trigger stream evaluates to a string value explaining the violation.

The sliding window counting values of the `rpm` input over the last 60 seconds is encoded using 60 bucket streams. Because counting does not depend on the actual values of the `rpm` stream, hold accesses to it are replaced with the constant 1. The aggregation then simply sums the bucket values to compute the total count of received values.

In the next chapter, we present a more extensive case study and compare the performance of these encodings with specifications that use a native implementation of periodic streams and sliding windows.

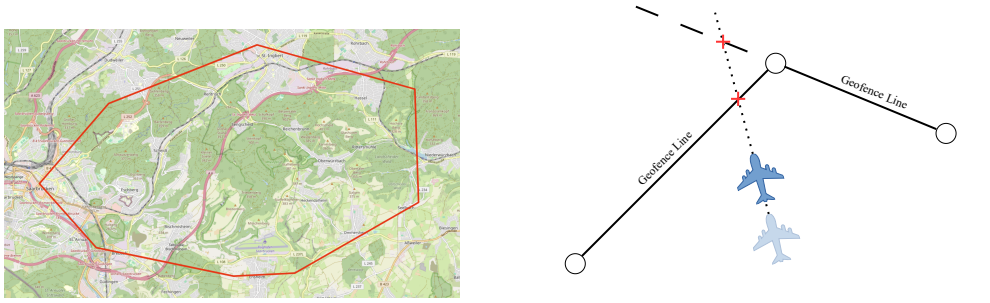
Chapter 5

A Case Study

In this chapter, we analyze the expressivity and performance of the fragment of RTLola specifications whose type safety can be guaranteed by StreamCore with conditional streams. We present two specifications from the aerospace domain in RTLola and their corresponding encodings in StreamCore. As an objective measure, we compare the analysis and interpretation performance of the specifications using the existing RTLola framework [Bau+24a].

5.1. Geofence

A geofence defines a virtual boundary for aircraft that must not be crossed to ensure safe flight and controlled airspace. In this example, the geofence is a polygon consisting of multiple lines forming its edges. For simplicity, the experiments use a geofence formed by four lines. A geographical visualization is given in Fig. 5.1a.



(a) A Visualization of a Geofence over the Saarbrücken Airport.

(b) A Plot Showing the Intersection of the Flight Path with the Geofence.

Figure 5.1.: A Visualization of a Geofence Specification.

To monitor such a geofence, the monitor observes the aircraft position via the `x` and `y` coordinates. It also observes the `velocity` to estimate the time until a potential breach. The calculation of the breach point is illustrated in Fig. 5.1b. Based on the previous and current aircraft positions, the trajectory is projected as a line. The monitor then computes intersection points between this trajectory and each geofence edge. If an intersection lies between the two vertices defining the edge, it is considered valid, and the distance from the aircraft to this point is computed. The minimal such distance indicates the projected breach location if the aircraft continues on a straight path. Using this distance and the aircraft's velocity, the time to breach is estimated. A RTLola specification for a monitor with a single geofence line is given in Spec. 5.1.

```
input x : Float
input y : Float
input velocity : Float

// Knots to m/s
output velocity_ms @velocity := velocity * 0.514444
constant c_epsilon : Float := 0.0000001

// Computes the vehicle line
output diff_x @x := x - x.offset(by: -1).defaults(to: x)
output diff_y @y := y - y.offset(by: -1).defaults(to: y)
output isFnc @x := diff_x != 0.0
output m @x && y:= if isFnc then (diff_y) / (diff_x) else 0.0
output b @x && y:= if isFnc then y-(m*x) else 0.0
output dstToPnt @x && y := sqrt(diff_x**2.0 + diff_y**2.0)

// true -> going into negative x direction; false -> positive x
output o_x @x := if abs(diff_x) < c_epsilon then false else diff_x < 0.0
output o_y @x&&y := if abs(diff_y) < c_epsilon then false else diff_y < 0.0

// Face 0
constant p0_x_0 :Float := 85.63214254449431
constant p0_y_0 :Float := 86.88834481791841
constant p1_x_0 :Float := 93.54204321854458
constant p1_y_0 :Float := 38.639226576496

output m_line_0 @x && y := (p1_y_0 - p0_y_0) / (p1_x_0 - p0_x_0)
output b_line_0 @x && y := p1_y_0 - (m_line_0 * p1_x_0)
output intersecting_0 @x && y := m != m_line_0 || b = b_line_0
output intersection_x_0 eval @x && y when isFnc && intersecting_0 with (b - b_line_0)
    / ( m_line_0 - m)
output intersection_y_0 eval @x && y when isFnc && intersecting_0 with m_line_0 *
    intersection_x_0+ b_line_0
output inbounds_0 eval @x && y when isFnc && intersecting_0 with
      if p0_x_0 < p1_x_0
      then (intersection_x_0 >= p0_x_0 and intersection_x_0 <= p1_x_0)
      else (intersection_x_0 <= p0_x_0 and intersection_x_0 >= p1_x_0)
output in_direction_0
```

```
37    eval @x && y when isFnc && intersecting_0 with if o_x then intersection_x_0 < x
          else intersection_x_0 > x
38  output violations_0
39    eval @x && y when isFnc && intersecting_0 && in_direction_0 && inbounds_0 with
          (intersection_x_0, intersection_y_0)
40  output distance_to_violations_0
41    eval @x && y when isFnc && intersecting_0 && in_direction_0 && inbounds_0 with
          sqrt((intersection_x_0 - x)**2.0 + (intersection_y_0 - y)**2.0)
42  output time_to_0
43    eval @(x && y) || velocity when isFnc.hold(or: false) && intersecting_0.hold(or:
          false) && in_direction_0.hold(or: false) && inbounds_0.hold(or: false) &&
          velocity_ms.hold(or: 0.0) != 0.0 with distance_to_violations_0.hold(or: 0.0) /
          velocity_ms.hold(or: 0.0)
```

Specification 5.1: A Geofence Specification.

This specification relies heavily on conditional streams, for example, to ensure that the trajectory can be expressed as $m * x + b$ and to avoid division-by-zero errors in the time-to-breach calculation. The StreamCore encoding is given in Appendix A.2. In this encoding, the `when` conditions are simplified compared to the RTLola specification, as they are implicitly strengthened by referencing other conditional streams in the pacing annotation instead of restating the evaluation condition of the accessed streams in the condition of the stream itself.

5.2. Tube

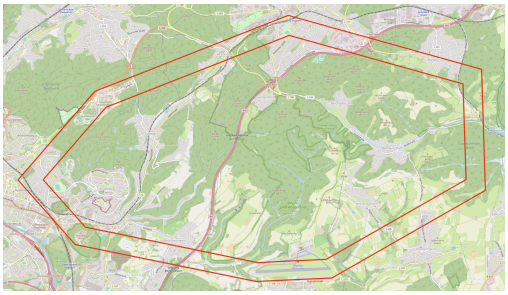
A tube is a stricter variant of a geofence, restricting deviations from a target flight path by a specified threshold. Unlike geofences, tubes are three-dimensional and also enforce adherence to the intended flight altitude. A top-down view is shown in Fig. 5.2a. Nevertheless, deviations from the intended path may be necessary, for example, to avoid collisions. Thus, it is often desirable to relax the tube requirement by allowing temporary deviations.

A tube monitor observes the `x`, `y`, and `z` positions of the aircraft and computes the distance to the closest flight path segment. The calculation for a single segment is illustrated in Fig. 5.2b. The monitor evaluates three distances:

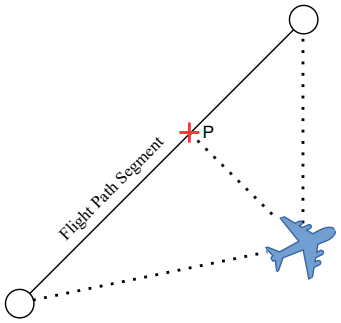
1. from the aircraft to the starting point of the segment,
2. from the current position to its projection on the segment (perpendicular distance),
3. from the aircraft to the endpoint of the segment.

The minimum of these distances is the distance between the aircraft and the segment.

The minimal distance to any segment defines the current distance to the path. If this exceeds the threshold, it constitutes a violation. For the relaxation, the monitor first issues a warning if the threshold is exceeded and only declares a violation if the condition persists for a specified duration. A RTLola specification for monitoring a single path fragment is given in Spec. 5.2.



(a) A Visualization of a Tube around a given Flight Path.



(b) A Plot Showing how a Tube can be Monitored.

Figure 5.2.: A Visualization of a Tube Specification.

```
input x : Float
input y : Float
input z : Float

constant threshold: Float := 350.0

// === Segment 0 ===
constant x0_0 : Float := 1.27
constant y0_0 : Float := 3.42
constant z0_0 : Float := 7.50

constant x1_0 : Float := 5.57
constant y1_0 : Float := 1.23
constant z1_0 : Float := 8.54

output output_ld0 @True := (x1_0 - x0_0, y1_0 - y0_0, z1_0 - z0_0)
output output_lc0 @x && y && z := (x - x0_0, y - y0_0, z - z0_0)
output output_lfp_len0 @x && y && z :=
  (output_ld0.0*output_lc0.0 + output_ld0.1*output_lc0.1 + output_ld0.2*output_lc0.2) /
  (output_ld0.0**2.0 + output_ld0.1**2.0 + output_ld0.2**2.0)
output output_lfp_proj0 @x && y && z :=
  (output_lfp_len0*output_ld0.0, output_lfp_len0*output_ld0.1,
      output_lfp_len0*output_ld0.2)
output output_lfp0 @x && y && z :=
  (x0_0 + output_lfp_proj0.0, y0_0 + output_lfp_proj0.1, z0_0 + output_lfp_proj0.2)
output output_d_lfp0 @x && y && z :=
  sqrt((output_lfp0.0 - x)**2.0 + (output_lfp0.1 - y)**2.0 + (output_lfp0.2 - z)**2.0)
output output_d_start0 @x && y && z :=
  sqrt((x0_0 - x)**2.0 + (y0_0 - y)**2.0 + (z0_0 - z)**2.0)
output output_d_end0 @x && y && z :=
  sqrt((x1_0 - x)**2.0 + (y1_0 - y)**2.0 + (z1_0 - z)**2.0)
```

```
output output_d_line0 @x && y && z :=
  if output_lfp_len0 <= 0.0 then output_d_start0
  else if output_lfp_len0 >= 1.0 then output_d_end0
  else output_d_lfp0

// Aggregate minimum distances
output min_distance @x && y && z := min(output_d_line0, ... ,output_d_linen)
output violated @x && y && z := min_distance > threshold
output critical_violation @2Hz := violated.aggregate(over:25s, using:forall)

trigger @x && y && z violated "warning : deviating from intended flight path!"
trigger @2Hz critical_violation "critical: left flight path. Return immediately!"
```

Specification 5.2: A Tube Specification

While this specification does not use conditional streams, it employs periodic streams and sliding window aggregations to implement the relaxation property. As a result, the encoding in StreamCore (given in Appendix A.3) grows significantly in size.

5.3. Evaluation

Although the contribution of this thesis is primarily theoretical, we present an evaluation based on the above specifications and the one from Sect. 1.1.3, whose encoding is given in Appendix A.1. We compare the specification length, analysis time, and runtime performance of the original RTLola specifications with their StreamCore encodings to assess the cost of ensuring type safety.

The specifications were analyzed using the existing RTLola Frontend [25b] and the RTLola Interpreter [24]. The StreamCore encodings were translated back into RTLola syntax while preserving the transformations. Timing measurements were performed with Criterion [25a], averaged over 20 randomly sampled runs. The experiments were executed on a PC with an 8-core processor at 4.5GHz and 32GB of memory. Traces of length 1000 were randomly sampled, ensuring the assumption on the `clock` input stream for the StreamCore specifications was satisfied. The results are shown in Tbl. 5.3.

Specification	**RTLola**			**StreamCore**		
	LOC	Analyze	Interpret	LOC	Analyze	Interpret
Geofence	149	217.0 ms	8.7 ms	149	199.1 ms	10.6 ms
Tube	142	404.7 ms	19.6 ms	208	4141.1 ms	43.4 ms
Motivating	15	2.6 ms	4.0 ms	160	10477.0 ms	24.6 ms

Table 5.3.: Comparison between LOC, Analyze and Interpretation Time for RTLola and StreamCore Variants of the Specifications.

As expected, encoding sliding windows as streams significantly increases specification size, especially for the *Tube* and *Motivating* examples (the latter referring to Sect. 1.1.3). The increased size also impacts analysis performance; for instance, the *Motivating* example requires 10.5 seconds compared to 3 milliseconds originally. Analysis time includes all frontend stages: parsing, type analysis, and well-definedness checking. Note that the RTLola framework is not optimized for such encodings, and a dedicated implementation of the StreamCore type system could be more efficient. Interpretation time also increases, due to the large number of streams introduced to encode sliding window operations. While the native RTLola Interpreter is at least twice as fast for sliding windows, neither its sliding window implementation nor its pacing annotations are proven type safe, which is the key contribution of this thesis.

Chapter 6

Conclusion and Future Work

This thesis establishes type safety for conditional streams in the stream-based monitoring language RTLola. Conditional streams extend the expressivity of the language, enabling safe formulations of properties that prevent errors such as division by zero. Building on this foundation, the thesis introduces an encoding of the real-time features of RTLola using conditional streams. These features include fixed-rate streams, encoded via a single clock input stream with a guaranteed frequency, and sliding window operations. Fixed-rate streams are particularly useful for detecting failed sensors or other components that no longer produce observable events.

To establish type safety for these features, the existing formalization of RTLola, called StreamCore, was extended from purely asynchronous streams to asynchronous streams with evaluation conditions. To preserve safety, that is, to guarantee that a model of the specification always exists regardless of input timing, the pacing type system was extended to allow output streams as references in pacing annotations. Consequently, the logical relations were revised, and a formal proof was developed. The proof shows that if a specification is well typed under the given inference rules, then it can be monitored without runtime errors caused by asynchrony or evaluation conditions.

On this basis, encodings of periodic streams and sliding window operations were defined in StreamCore using conditional streams. Periodic streams rely on a single clocked input stream, while their scheduling relative to the clock is expressed as an evaluation condition. Sliding window aggregations are encoded as sequences of periodic streams that replicate the bucketing algorithm of sliding window operations.

The expressivity and runtime performance of this fragment was evaluated in a small-scale case study with specifications from the aerospace domain. As expected, performance measurements show that the native implementation of sliding windows outperforms their encoded counterpart. Nevertheless, the study demonstrates that the fragment of RTLola with proven type safety is expressive enough to capture complex properties, such as approximate adherence to a given flight path.

6.1. Future Work

Several extensions of this work are possible. First, an implementation of a type checker for the presented system is required. Although the RTLola framework already includes a type checker for specifications with evaluation clauses and periodic streams, its correctness is not formally proven. A comparison between the existing type checker and a new implementation based on the soundness results presented here would be of particular interest.

Second, as discussed in [Koh+25], the constructive nature of the proof allows for the automatic derivation of a verified interpreter for StreamCore. Such an interpreter would be valuable for safety-critical cyber-physical systems. It could also be used for differential testing against the optimized RTLola implementation to improve robustness.

Third, several language features of RTLola remain without proven type safety. One such feature is dynamic streams. Dynamic streams are created and terminated during execution using the `spawn` and `close` expressions, which affect specification safety in a manner similar to evaluation conditions. In many cases, spawn and close conditions could be encoded using evaluation conditions. An exception arises when dynamically created streams are periodic. In such cases, the local clock of the stream may be misaligned with other streams of the same frequency, as its clock begins only at creation time. Local periodic streams are important for specifying properties such as deadline conformance, where a property must hold within a fixed duration after an event [Bau+24a].

Another feature not yet captured by StreamCore with conditional streams is parameterized streams. Parameterized streams generalize output streams to sets of stream instances, each created and closed by spawn and close expressions. Safety issues arise when these instances access each other synchronously. Parameterized streams are especially useful for representing unbounded data domains, such as monitoring an arbitrary number of surrounding aircraft.

In summary, further formalization is required to ensure the safe evaluation of these expressive language features.

Appendix A

Case-Study Specifications encoded in StreamCore

This appendix contains the translated versions of the RTLola specifications presented in the case-study in Chapter 5 into StreamCore with conditional streams.

A.1. Running Example from Sect. 1.1.3

The StreamCore version of the specification presented in Sect. 1.1.3.

```
1  input rpm: Float
2  input position: Float
3  // Assume to tick at a frequency of 1Hz
4  input clock: Bool
5
6  output step @clock := (step.prev(0) + 1) % 1
7
8  // Constructed clock streams
9  output clock_1Hz @clock when step % 1 == 0 := true
10
11 output velocity @position := (position - position.prev(position)) * 36.0
12 output trigger_too_fast @position when velocity > 150.0 := "Driving too fast"
13
14 // Sliding window buckets for count
15 output count_b_60 @clock_1Hz ∨ rpm :=
16       if clock_1Hz.fresh() then
17          if rpm.fresh() then
18             1
19          else
20             0
21       else
22          count_b_60.prev(0) + 1
```

```
output count_b_59 @clock_1Hz := count_b_60.prev(0)
       ⋮
output count_b_2 @clock_1Hz := count_b_3.prev(0)
output count_b_1 @clock_1Hz := count_b_2.prev(0)

output count @clock_1Hz:= count_b_1.prev(0) + ... + count_b_60.prev(0)

// Sliding window buckets for sum
output sum_b_60 @clock_1Hz ∨ rpm :=
      if clock_1Hz.fresh() then
         if rpm.fresh() then
            rpm.hold(0.0)
         else
            0.0
      else
         sum_b_60.prev(0.0) + rpm.hold(0.0)
output sum_b_59 @clock_1Hz := sum_b_60.prev(0.0)
       ⋮
output sum_b_2 @clock_1Hz := sum_b_3.prev(0.0)
output sum_b_1 @clock_1Hz := sum_b_2.prev(0.0)

output sum @clock_1Hz:= sum_b_1.prev(0) + ... + sum_b_60.prev(0)

output avg_rpm @clock_1Hz when count > 0 := sum / count
output trigger_sensor_fail @clock_1Hz when count < 60 := "The RPM sensor seems to have
    failed"

output shift @position || rpm := velocity.hold(0.0) < 30.0 && avg_rpm.hold(0.0) >
    3000.0
output trigger_shift @position || rpm when shift := "Shift to higher gear"
```

A.2. A Geofence Specification

This section contains the geofence specification presented in Chapter 5 translated into StreamCore with conditional streams.

```
input x : Float
input y : Float
input velocity : Float

// Knots to m/s
output velocity_ms @velocity := velocity * 0.514444
output c_epsilon @True := 0.0000001

// Computes the vehicle line
output diff_x @x := x - x.prev(x)
output diff_y @y := y - y.prev(y)
output isFnc @x := diff_x != 0.0
```

```
output m @x && y:= if isFnc then (diff_y) / (diff_x) else 0.0
output b @x && y:= if isFnc then y-(m*x) else 0.0
output dstToPnt @x && y := sqrt(diff_x**2.0 + diff_y**2.0)

// true -> going into negative x direction; false -> positive x
output o_x @x := if abs(diff_x) < c_epsilon then false else diff_x < 0.0
output o_y @x&&y := if abs(diff_y) < c_epsilon then false else diff_y < 0.0

// Face 0
output p0_x_0 @True := 85.63214254449431
output p0_y_0 @True := 86.88834481791841
output p1_x_0 @True := 93.54204321854458
output p1_y_0 @True := 38.639226576496

output m_line_0 @x && y := (p1_y_0 - p0_y_0) / (p1_x_0 - p0_x_0)
output b_line_0 @x && y := p1_y_0 - (m_line_0 * p1_x_0)
output intersecting_0 @x && y := m != m_line_0 || b = b_line_0
output intersection_x_0 @x && y when isFnc && intersecting_0 := (b - b_line_0) / (
    m_line_0 - m)
output intersection_y_0 @intersection_x_0 := m_line_0 * intersection_x_0 + b_line_0
output inbounds_0 @intersection_x_0 := if p0_x_0 < p1_x_0
                      then (intersection_x_0 >= p0_x_0 and intersection_x_0 <= p1_x_0)
                      else (intersection_x_0 <= p0_x_0 and intersection_x_0 >= p1_x_0)
output in_direction_0 @intersection_x_0 := if o_x then intersection_x_0 < x else
    intersection_x_0 > x
output violations_0 @intersection_x_0 when in_direction_0 && inbounds_0 :=
    (intersection_x_0, intersection_y_0)
output distance_to_violations_0 @intersection_x_0 when in_direction_0 && inbounds_0 :=
    sqrt((intersection_x_0 - x)**2.0 + (intersection_y_0 - y)**2.0)
output time_to_0
   eval @distance_to_violations_0 || velocity when velocity_ms.hold(or: 0.0) != 0.0
       with distance_to_violations_0.hold(or: 0.0) / velocity_ms.hold(or: 0.0)
```

A.3. A Tube Specification

This section contains the tube specification presented in Chapter 5 translated into StreamCore with conditional streams.

```
input x : Float
input y : Float
input z : Float

// Assume to tick at a frequency of 2Hz
input clock: Bool

output step @clock := (step.prev(0) + 1) % 1

// Constructed clock streams
output clock_2Hz @clock when step % 1 == 0 := true
```

```
12
13 output threshold @True := 350.0
14
15 // === Segment 0 ===
16 output x0_0 @True := 1.27
17 output y0_0 @True := 3.42
18 output z0_0 @True := 7.50
19
20 output x1_0 @True := 5.57
21 output y1_0 @True := 1.23
22 output z1_0 @True := 8.54
23
24 output output_ld0 @x := (x1_0 - x0_0, y1_0 - y0_0, z1_0 - z0_0)
25 output output_lc0 @x&&y&&z := (x - x0_0, y - y0_0, z - z0_0)
26 output output_lfp_len0 @x&&y&&z :=
27   ((output_ld0.0*output_lc0.0 + output_ld0.1*output_lc0.1 + output_ld0.2*output_lc0.2) /
28   (output_ld0.0**2.0 + output_ld0.1**2.0 + output_ld0.2**2.0))
29 output output_lfp_proj0 @x&&y&&z :=
30   (output_lfp_len0*output_ld0.0, output_lfp_len0*output_ld0.1,
       output_lfp_len0*output_ld0.2)
31 output output_lfp0 @x&&y&&z :=
32   (x0_0 + output_lfp_proj0.0, y0_0 + output_lfp_proj0.1, z0_0 + output_lfp_proj0.2)
33 output output_d_lfp0 @x&&y&&z :=
34   sqrt((output_lfp0.0 - x)**2.0 + (output_lfp0.1 - y)**2.0 + (output_lfp0.2 - z)**2.0)
35 output output_d_start0 @x&&y&&z :=
36   sqrt((x0_0 - x)**2.0 + (y0_0 - y)**2.0 + (z0_0 - z)**2.0)
37 output output_d_end0 @x&&y&&z :=
38   sqrt((x1_0 - x)**2.0 + (y1_0 - y)**2.0 + (z1_0 - z)**2.0)
39 output output_d_line0 @x&&y&&z :=
40   if output_lfp_len0 <= 0.0 then output_d_start0
41   else if output_lfp_len0 >= 1.0 then output_d_end0
42   else output_d_lfp0
43
44 // Aggregate minimum distance
45 output min_distance @x&&y&&z := min(output_d_line0, ... ,output_d_line_n)
46 output violated @x&&y&&z := min_distance > threshold
47
48 // Sliding window buckets for long_violation
49 output b_50 @(clock_2Hz || (x&&y&&z)) :=
50    if clock_2Hz.fresh() then
51      if violated.fresh() then
52             violated.hold(false)
53          else
54             false
55       else
56          b_50.prev(false) && violated.hold(false)
57 output b_49 @clock_2Hz := b_50.prev(false)
58 output b_48 @clock_2Hz := b_49.prev(false)
59       ⋮
60 output b_1 @clock_2Hz := b_2.prev(false)
```

```
61
62 output long_violation @clock_2Hz := b_1.prev(false) && ... && b_50.prev(false)
63
64 output trigger_warning @x&&y&&z when violated := "warning : deviating from intended
       flight path!"
65 output trigger_critical @clock_2Hz when long_violation := "critical: left flight path.
       Return immediately!"
```

Bibliography

[24] *RTLola Interpreter - Rust Crate on crates.io*. 2024. URL: https://crates.io/crates/rtlola-interpreter (visited on 09/26/2025).

[25a] *Criterion - Statistics-driven micro-benchmarking library*. 2025. URL: https://crates.io/crates/criterion (visited on 09/26/2025).

[25b] *RTLola Frontend - Rust Crate on crates.io*. 2025. URL: https://crates.io/crates/rtlola-frontend (visited on 09/26/2025).

[Bau+20a] Jan Baumeister et al. "Automatic Optimizations for Stream-Based Monitoring Languages". In: *Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings*. Ed. by Jyotirmoy Deshmukh and Dejan Nickovic. Vol. 12399. Lecture Notes in Computer Science. Springer, 2020, pp. 451–461. DOI: 10.1007/978-3-030-60508-7_25.

[Bau+20b] Jan Baumeister et al. "RTLola Cleared for Take-Off: Monitoring Autonomous Aircraft". In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12225. Lecture Notes in Computer Science. Springer, 2020, pp. 28–39. DOI: 10.1007/978-3-030-53291-8_3.

[Bau+24a] Jan Baumeister et al. "A Tutorial on Stream-Based Monitoring". In: *Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part II*. Ed. by André Platzer et al. Vol. 14934. Lecture Notes in Computer Science. Springer, 2024, pp. 624–648. DOI: 10.1007/978-3-031-71177-0_33. URL: https://doi.org/10.1007/978-3-031-71177-0%5C_33.

[Bau+24b] Jan Baumeister et al. "Monitoring Unmanned Aircraft: Specification, Integration, and Lessons-Learned". In: *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II*. Ed. by Arie Gurfinkel and Vijay Ganesh. Vol. 14682. Lecture

Notes in Computer Science. Springer, 2024, pp. 207–218. DOI: 10.1007/978-3-031-65630-9_10. URL: https://doi.org/10.1007/978-3-031-65630-9%5C_10.

[Bau+25] Jan Baumeister et al. "Stream-Based Monitoring of Algorithmic Fairness". In: *Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part I*. Ed. by Arie Gurfinkel and Marijn Heule. Vol. 15696. Lecture Notes in Computer Science. Springer, 2025, pp. 60–81. DOI: 10.1007/978-3-031-90643-5_4. URL: https://doi.org/10.1007/978-3-031-90643-5%5C_4.

[BC84] Gérard Berry and Laurent Cosserat. "The ESTEREL Synchronous Programming Language and its Mathematical Semantics". In: *Seminar on Concurrency, Carnegie-Mellon University, Pittsburg, PA, USA, July 9-11, 1984*. Ed. by Stephen D. Brookes, A. W. Roscoe, and Glynn Winskel. Vol. 197. Lecture Notes in Computer Science. Springer, 1984, pp. 389–448. DOI: 10.1007/3-540-15670-4_19.

[Ben+11] Albert Benveniste et al. "A hybrid synchronous language with hierarchical automata: static typing and translation to synchronous code". In: *Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*. Ed. by Samarjit Chakraborty et al. ACM, 2011, pp. 137–148. DOI: 10.1145/2038642.2038664. URL: https://doi.org/10.1145/2038642.2038664.

[Ben+14] Albert Benveniste et al. "A type-based analysis of causality loops in hybrid systems modelers". In: *17th International Conference on Hybrid Systems: Computation and Control (part of CPS Week), HSCC'14, Berlin, Germany, April 15-17, 2014*. Ed. by Martin Fränzle and John Lygeros. ACM, 2014, pp. 71–82. DOI: 10.1145/2562059.2562125. URL: https://doi.org/10.1145/2562059.2562125.

[BKZ17] David A. Basin, Felix Klaedtke, and Eugen Zalinescu. "Runtime Verification of Temporal Properties over Out-of-Order Data Streams". In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*. Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10426. Lecture Notes in Computer Science. Springer, 2017, pp. 356–376. DOI: 10.1007/978-3-319-63387-9_18.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Runtime verification for LTL and TLTL". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20.4 (2011), pp. 1–64.

[Cas+87] Paul Caspi et al. "Lustre: A Declarative Language for Programming Synchronous Systems". In: *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*. ACM Press, 1987, pp. 178–188. DOI: 10.1145/41625.41641.

[Che+24] Jiawei Chen et al. "Synchronous Programming with Refinement Types". In: *Proc. ACM Program. Lang.* 8.ICFP (2024), pp. 938–972. DOI: 10.1145/3674657. URL: https://doi.org/10.1145/3674657.

[CP03] Jean-Louis Colaço and Marc Pouzet. "Clocks as First Class Abstract Types". In: *Embedded Software, Third International Conference, EMSOFT 2003, Philadelphia, PA, USA, October 13-15, 2003, Proceedings*. Ed. by Rajeev Alur and Insup Lee. Vol. 2855. Lecture Notes in Computer Science. Springer, 2003, pp. 134–155. DOI: 10.1007/978-3-540-45212-6_10. URL: https://doi.org/10.1007/978-3-540-45212-6%5C_10.

[dAn+05] Ben d'Angelo et al. "LOLA: runtime monitoring of synchronous systems". In: *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*. IEEE. 2005, pp. 166–174.

[DGP08] Gwenaël Delaval, Alain Girault, and Marc Pouzet. "A type system for the automatic distribution of higher-order synchronous dataflow programs". In: *Proceedings of the 2008 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08), Tucson, AZ, USA, June 12-13, 2008*. Ed. by Krisztián Flautner and John Regehr. ACM, 2008, pp. 101–110. DOI: 10.1145/1375657.1375672. URL: https://doi.org/10.1145/1375657.1375672.

[Fay+19] Peter Faymonville et al. "StreamLAB: Stream-based Monitoring of Cyber-Physical Systems". In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019, pp. 421–431. DOI: 10.1007/978-3-030-25540-4_24.

[GL87] Thierry Gautier and Paul Le Guernic. "SIGNAL: A declarative language for synchronous programming of real-time systems". In: *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings*. Ed. by Gilles Kahn. Vol. 274. Lecture Notes in Computer Science. Springer, 1987, pp. 257–277. DOI: 10.1007/3-540-18317-5_15. URL: https://doi.org/10.1007/3-540-18317-5%5C_15.

[GS21] Felipe Gorostiaga and César Sánchez. "HStriver: A Very Functional Extensible Tool for the Runtime Verification of Real-Time Event Streams". In: *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*. Ed. by Marieke Huisman, Corina S. Pasareanu, and

Naijun Zhan. Vol. 13047. Lecture Notes in Computer Science. Springer, 2021, pp. 563–580. DOI: 10.1007/978-3-030-90870-6_30.

[Hal+91] Nicolas Halbwachs et al. "The synchronous data flow programming language LUSTRE". In: *Proc. IEEE* 79.9 (1991), pp. 1305–1320. DOI: 10.1109/5.97300.

[Hen+23] Thomas A. Henzinger et al. "Monitoring Algorithmic Fairness". In: *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*. Ed. by Constantin Enea and Akash Lal. Vol. 13965. Lecture Notes in Computer Science. Springer, 2023, pp. 358–382. DOI: 10.1007/978-3-031-37703-7_17.

[Hin69] Roger Hindley. "The principal type-scheme of an object in combinatory logic". In: *Transactions of the american mathematical society* 146 (1969), pp. 29–60.

[Jin+12] Dongyun Jin et al. "JavaMOP: Efficient parametric runtime monitoring framework". In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. Ed. by Martin Glinz, Gail C. Murphy, and Mauro Pezzè. IEEE Computer Society, 2012, pp. 1427–1430. DOI: 10.1109/ICSE.2012.6227231. URL: https://doi.org/10.1109/ICSE.2012.6227231.

[JTS21] Sebastian Junges, Hazem Torfah, and Sanjit A. Seshia. "Runtime Monitors for Markov Decision Processes". In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12760. Lecture Notes in Computer Science. Springer, 2021, pp. 553–576. DOI: 10.1007/978-3-030-81688-9_26.

[Kal+22] Hannes Kallwies et al. "TeSSLa - An Ecosystem for Runtime Verification". In: *Runtime Verification - 22nd International Conference, RV 2022, Tbilisi, Georgia, September 28-30, 2022, Proceedings*. Ed. by Thao Dang and Volker Stolz. Vol. 13498. Lecture Notes in Computer Science. Springer, 2022, pp. 314–324. DOI: 10.1007/978-3-031-17196-3_20.

[Kim+99] Moonjoo Kim et al. "Formally specified monitoring of temporal properties". In: *11th Euromicro Conference on Real-Time Systems (ECRTS 1999), 9-11 June 1999, York, England, UK, Proceedings*. IEEE Computer Society, 1999, pp. 114–122. DOI: 10.1109/EMRTS.1999.777457. URL: https://doi.org/10.1109/EMRTS.1999.777457.

[Koh+25] Florian Kohn et al. *Pacing Types: Safe Monitoring of Asynchronous Streams*. 2025. arXiv: 2509.06724 [cs.PL]. URL: https://arxiv.org/abs/2509.06724.

[Koy90] Ron Koymans. "Specifying Real-Time Properties with Metric Temporal Logic". In: *Real Time Syst.* 2.4 (1990), pp. 255–299. DOI: 10.1007/BF01995674. URL: https://doi.org/10.1007/BF01995674.

[Mil78] Robin Milner. "A Theory of Type Polymorphism in Programming". In: *J. Comput. Syst. Sci.* 17.3 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4. URL: https://doi.org/10.1016/0022-0000(78)90014-4.

[MPP10] Louis Mandel, Florence Plateau, and Marc Pouzet. "Lucy-n: a n-Synchronous Extension of Lustre". In: *Mathematics of Program Construction, 10th International Conference, MPC 2010, Québec City, Canada, June 21-23, 2010. Proceedings*. Ed. by Claude Bolduc, Josée Desharnais, and Béchir Ktari. Vol. 6120. Lecture Notes in Computer Science. Springer, 2010, pp. 288–309. DOI: 10.1007/978-3-642-13321-3_17. URL: https://doi.org/10.1007/978-3-642-13321-3%5C_17.

[PGD24] Ivan Perez, Alwyn E. Goodloe, and Frank Dedden. "Runtime Verification in Real-Time with the Copilot Language: A Tutorial". In: *Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part II*. Ed. by André Platzer et al. Vol. 14934. Lecture Notes in Computer Science. Springer, 2024, pp. 469–491. DOI: 10.1007/978-3-031-71177-0_27. URL: https://doi.org/10.1007/978-3-031-71177-0%5C_27.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN: 978-0-262-16209-8.

[Pnu77] Amir Pnueli. "The Temporal Logic of Programs". In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32. URL: https://doi.org/10.1109/SFCS.1977.32.

[Sch21] Frederik Scheerer. "Monitoring Smart Contracts with RTLola". In: *Bachelor's Thesis, Saarland University* (2021).

[Sko19] Lau Skorstengaard. *An Introduction to Logical Relations*. 2019. arXiv: 1907.11133 [cs.PL]. URL: https://arxiv.org/abs/1907.11133.

[Sta85] Richard Statman. "Logical Relations and the Typed lambda-Calculus". In: *Inf. Control.* 65.2/3 (1985), pp. 85–97. DOI: 10.1016/S0019-9958(85)80001-2. URL: https://doi.org/10.1016/S0019-9958(85)80001-2.