Let's not Trust Experience Blindly: Formal Monitoring of Humans and other CPS

Saarland University

Department of Computer Science

Master's Thesis

submitted by

Maximilian Schwenger

Saarbrücken, September 2019

Supervisor:	Prof. Bernd Finkbeiner, Ph. D.
Reviewer:	Prof. Bernd Finkbeiner, Ph. D. Prof. Dr. Jan Reinecke
Submission:	24. September 2019

Abstract

The control logic of complex systems is based on experience: Trained experts steer a machine directly until they help develop an automated controller. Recently, this process was further improved by successfully incorporating machine learning techniques, where the controller was learned from tremendous amounts of empirical data. The resulting controller excels most of the time, especially in situations similar to ones occurring in the training data. In a safety-critical context, however, this is not enough, so formal guarantees about the behavior of the controller become crucial. When a full static analysis and subsequent verification is infeasible due to the complexity of the system, runtime monitoring is still applicable. It acts as a connecting link between the efficiency of trained controllers and formally verifiable guarantees. A runtime monitor assesses the system health based on sensor readings by using a specification that contains information about desired system states and their expected evolution over time. When the monitor encounters a violation of the specification, it raises an alarm. For complex systems, characterizing the desired behavior requires an expressive language. Moreover, provably correct behavior requires formal semantics and an evaluation algorithm with static guarantees on resource consumption to prevent crashes during runtime. This thesis presents formal semantics for the specification language RTLola and shows that it satisfies the aforementioned criteria by introducing an evaluation algorithm with static time and space bounds. The approach is evaluated based on examples from health monitoring and aircraft controllers.

Zusammenfassung

Seit vielen Jahren basiert die Kontrolllogik von Systemen auf Erfahrung: Ausgebildete Experten steuern eine Maschine selbst bis sie bei der Entwicklung eines automatischen Reglers mithelfen. Neuerdings wird dieser Prozess weiter verbessert indem erfolgreiche Machine Learning Techniken zum Einsatz kommen, wobei ein Regler mit Hilfe enormer Mengen empirischer Daten gelernt wird. Der resultierender Regler liefert die meiste Zeit hervorragende Resultate, insbesondere in Situationen, die ähnlich zu bereits erfahrenen sind. In einem sicherheitskritischen Kontext ist dies jedoch nicht ausreichend, weshalb formale Garantien bezüglich des Verhaltens des Systems wesentlich werden. Wenn eine volle, statische Analyse und nachfolgende Verifikation jedoch aufgrund der Komplexität des Systems nicht praktikabel ist, kann Laufzeitüberwachung weiterhin anwendbar sein. Es agiert dabei als Bindeglied zwischen der Effizienz von empirisch trainierten Reglern und formal verifizierbaren Garantien. Eine Laufzeitüberwachung evaluiert den Zustand des Systems basierend auf Sensorwerten unter Berücksichtigung einer Spezifikation, die Informationen über wünschenswerte Systemzustände und deren erwartete Evolution über die Zeit enthält. Sobald die Überwachung eine Verletzung feststellt, wird ein Alarm ausgelöst. Für komplexe Systeme benötigt eine solche Charakterisierung jedoch eine expressive Sprache. Darüberhinaus verlangt beweisbar korrektes Verhalten eine formale Semantik und einen Evaluationsalgorithmus mit statischen Garantien bezüglich Ressourcenverbrauch. Diese Abschlussarbeit präsentiert eine formale Semantik für die Spezifikationssprache RTLola und zeigt, dass diese die zuvor genannten Kriterien erfüllt, indem sie einen Evaluationsalgorithmus mit statischen Laufzeit- und Speichergrenzen vorstellt. Der Ansatz wird mit Hilfe von Beispielen aus der Gesundheitsüberwachung und Avionik validiert.

Acknowledgements

First, I wish to thank my supervisor Bernd Finkbeiner for giving me a chance to write this thesis. Thanks to Jan Baumeister, Leander Tentrup, Malte Schledjewski, Marvin Stenger, and Hazem Torfah, as well as Sebastian Schirmer and Christoph Torens from the DLR for great discussions — without you, the project would certainly not be in its current state. Further, I want to thank the remainder of the Reactive Systems group for your great company, Julia Wichlacz for proofreading, and all my friends and family, especially my dearest sister Linda, for general support. Last but not least, thank you to Jan Reinecke for reviewing 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, 24. September 2019

Contents

Chapter 1

Introduction

A recurring theme in global development is to simplify complex tasks such that the labor can be shifted from experts to an expendable work force with little need for supervision. As an example, consider the first automobile, the Benz Patent-Motorwagen. The prototype was assembled by its inventor Carl Benz, and so were the following few models. Only when the assembly process was sufficiently well-understood, less skilled mechanics were entrusted with the task. Over the last 130 years, the processes was continually simplified and entrusted to low-paid assemblymen. Nowadays, the task is automated using robot arms up to a point where little to no manual intervention is necessary. Programming these arms, however, requires experience in two fields: knowledge about the construction process and a firm grasp on the capabilities and limitations of the arms. After deployment of the arms, when experience proved the automated process sufficiently stable, the amount of supervision is reduced to a minimum. In essence, experience allows the process to work almost-fully automated with very little supervision.

A similar development can be seen for users of automobiles. The first driving license[1] was specially issued to Carl Benz for his Motorwagen. Successor licenses required drivers to have knowledge about the mechanics of the vehicle. In contrast to this, current student drivers barely need to know how to refill hydraulic fluid. Similarly, in many countries, they are no longer required to train the operation of a car with manual transmission. And the trend continues: there is a steady shift from level 2 autonomy, i.e., advanced driving assistance allowing the driver to disengage from physically operating the car, towards level 3 autonomy, where the driver's necessity to monitor the situation ceases under normal circumstances. Increasing autonomy leads to fewer requirements on the operator, so it seems reasonable to lower the required training before granting driving licenses in the future.

[1] the "permit for conducting test drives in a patented engine vehicle" (German: "Berechtigung zur Durchführung von Versuchsfahrten mit einem Patentmotorwagen")

Yet again, the development process of autonomous cars heavily relies on experience: empirical testing reveals that the autopilot behaves correctly in all tested scenarios. Experience also plays a crucial role in the software itself, though less obvious. The controller software of the car uses machine learned neural nets for image recognition, prediction, and path planning. The training set for these nets consist of millions of driving kilometers, which is realistic empirical data, also known as "experience".

The learned controllers excel most of the time, often delivering vastly superior results compared to any existing tool. The disconcerting part about this statement is the adverb "often". The caveat about learned components is that they are highly cryptic; explaining their decision process is considered impossible. This is no surprise considering that experience is similarly hard to explain: An experienced driver intuitively knows to which position within their lane they should steer their car. However, they cannot justify precisely why this position is correct. So if they overshoot and crash, a post-mortem analysis often turns out inconclusive. As a bottomline, systems based on experience, be it human experience or a comprehensive data set, are well-behaved most of the time but come with flaws.

These flaws can have dire consequences when considering a second development over time: automated tasks become ever more safety-critical. The confidence in automated systems rose from regulating the water level of a water clock in 270 BC to controlling nuclear power plants in the 21st century.

Automated systems are deployed in a variety of critical areas such as in medical cyber-physical systems or in avionics. While they performed reasonable well in the past, several incidents show that misbehavior leads to significant ramifications. Between 1985-1987, for example, a data race in the Therac-25 radiation therapy machine lead to severe radiation poisoning in several patients, causing multiple cases of debilitation and three deaths.[2] More recently, in March 2019, the autopilot of a Tesla car failed to recognize a truck towing a trailer, which lead to a fatal accident.[3]

These incidents reinforce the demand for formal, static guarantees on the system. For this, a verifier analyses all possible executions of a system and verifies them against a specification of the desired behavior. This requires a model of the system which accurately represents the ramifications of a control decision. For purely discrete programs, an adequate model is easy to obtain. However, many safety-critical systems are deployed in a cyber-physical context. A cyber-physical system is a discrete computer program that interacts with the continuous, physical world. Examples are autonomous cars and aircraft, humans with medical implants, or robotic arms in factories.

The physical world is so chaotic that a precise deterministic model is impossible to find [1]. Moreover, the complexity of the physical world renders even imprecise models large. As a result, finding a model adequately representing the system often leads to a verification task that does not scale sufficiently well. The main reason behind this is that static verifiers check every possible execution. In contrast to that, runtime verification is only concerned with a single execution trace. Thus, the applicability of runtime

[2] https://www.bugsnag.com/blog/bug-day-race-condition-therac-25
[3] https://www.wired.com/story/teslas-latest-autopilot-death-looks-like-prior-crash/

verification significantly exceeds static verification. The basic idea of runtime verification is to deploy a dedicated component, the monitor, on the system under scrutiny. The monitor assesses the state of the system based on sensor readings and analyzes it w.r.t. a formal specification. When detecting a violation, the runtime monitor reports the problem to the controller, which initiates counter measures. Such counter measures can be switching from the highly complex, effective and efficient controller to a simple, verified safety-controller. In an autonomous taxi, for example, the safety-controller can decelerate and pull over. This strategy is safe albeit ineffective w.r.t. its goal of transporting passengers to a target location.

Runtime monitoring thus acts as a connection link between the efficacy of the empirically trained controllers and formally verifiable guarantees. For granting guarantees, however, the correctness of the monitor itself needs to be formally verified, and the specification needs to represent the desired behavior of the system accurately. This combination proves problematic for many runtime verification approaches because the specification language is a compromise between formal guarantees and expressiveness. Consider, for example, linear-time logic (LTL) [2]. The logic is studied extensively and thus well understood. Respective monitors [3] are based on finite state automata, with constant memory and performance overhead. However, what the logic has in static guarantees, it lacks in expressiveness. Intuitively, LTL cannot count, so encoding requirements such as "the location approximation based on the GPS module and the IMU shall not diverge by more than 50 meters" is a hassle or even impossible. More expressive logics come with the drawback of a non-constant memory and runtime overhead in the length of the trace. As a result, the longer a system is deployed, the more time and space it takes to verify compliance with the specification. Another way to specify properties is using general purpose programming languages. They are sufficiently expressive and the engineers developing the system are already familiar with them, making the integration easier. However, many programming languages do not feature formal semantics, which are required for formal guarantees. Moreover, a static analysis of such a monitor becomes hard due to the expressiveness of the language.

The prototype language RTLola presented by Faymonville [4] aims to solve this problem by providing a set of common operations as language primitives. These operations are selected to provide static guarantees regarding their runtime overhead. This thesis builds upon this by presenting a version of the language with overhauled and extended syntax. Moreover, we formally define the semantics of the language and present a monitoring algorithm that provably complies to the semantics. The resulting monitor requires a statically bounded amount of memory and only imposes a constant amount of runtime overhead.

The approach was implemented in a tool with the name StreamLab that generates a monitor with the aforementioned features out of an RTLola specification. We validate the tool and language with two case studies, based on health monitoring and aircraft controllers.

1.1. RTLola by Example

The best way to introduce a language like RTLola is on an intuitive level. The next sections will then formalize the semantics and show how to efficiently monitor an RTLola specification.

An RTLola specification consists of stream declarations. Each of them declares either an *input stream*, an *output stream*, or a *trigger*. The system collects information such as sensor readings, and passes it on to the monitor. Each data packet is mapped to one of the declared input streams. Output streams then access input data, aggregate and refine until it is useable to assess the health state of a system.

As a running example consider a car with an accelerometer measuring the acceleration in driving direction. The car shall drive smoothly, so it may never accelerate or decelerate with more than 4m/s^2. An RTLola specification for this declares an input stream for the measurements of the accelerometer, and an output stream checking whether the absolute acceleration exceeds 4. If so, a trigger sends a message to the controller of the car.

```
input accel_mpss: Float64
output high_accel: Bool := abs(accel_mpss) > 4
trigger high_accel "Ride not smooth."
```

Note that the suffix of stream name indicates the unit; this is not part of RTLola but a common convention. Note that RTLola is a strongly typed language and requires type annotations for input streams. Output stream types, however, can often be inferred and are thus optional most of the time.

Assume the aforementioned care should additionally ensure that the speed limit of 50km/h is respected. Recall that the acceleration is the first derivative of the velocity and the second derivative of the distance. As a result, integrating the acceleration yields the velocity. RTLola provides primitives for involved common operations such as aggregations over a pre-defined history of values. This includes the time-bounded integration of input stream values. Thus, we declare an output stream computing the difference in velocity Δv over the last second by integrating all readings of the accelerometer in this time frame. Then, the overall velocity v is the sum of all Δv values. Expressed as a formula, the specification computes the following for a point in time $t \in \mathbb{N}$:

$$v(t) = \int_0^t \dot{v}(\tau)d\tau = \int_0^t a(\tau)d\tau = \sum_{i=0}^{t-1}\left(\int_i^{i+1} a(\tau)d\tau\right) = \sum_{i=0}^{t-1} \Delta v_i \tag{1.1}$$

In RTLola, such a specification looks as follows:

```
input accel_mpss: Float64
output high_accel: Bool := abs(accel_mpss) > 4
trigger high_accel "Ride not smooth."

output Δv_mps: Float64 @1Hz := accel.aggregate(over: 1s, using: ∫)
output v_mps: Float64 @1Hz := v_mps.offset(by: -1).defaults(to: 0.0) + Δv_mps
```

```
output v_kmph: Float64 := v_mps * 3.6
trigger v_kmph > 50 "Driving faster than permitted."
```

The specification exhibits several noteworthy characteristics. The Δv_mps stream declares an extension frequency of 1Hz. This prompts the monitor to compute Δv_mps once per second only, rendering Δv_mps a *periodic stream*. In contrast to that, high_accel in the last specification was *event-based*, i.e., the monitor computed its value whenever the accelerometer generated a new reading. The detailed rationale behind periodic streams will be explained in Section 5.2. For now, the only relevant information is that an output with a *sliding window* expression is required to be periodic. A sliding window is a time frame of a fixed length, in this case 1s, whose values get aggregated using a specified function. In the case, the aggregation function is the integral.

→ Sec. 5.2, Page 57

Δv_mps only takes the "recent" acceleration data into account. For the total velocity, v_mps sums up all values of Δv_mps. The stream refers to itself with an *offset* of -1, meaning it takes the last computed value of itself. If this value does not exists — which is the case when the monitor just started — it falls back to the specified default value 0. Thus, the stream actively holds a state. This distinguished RTLola from many rule-based specification languages like RuleR [5] or Snort [6].

The last major characteristic of RTLola covered in this brief introduction is its extensibility and modularity. The former specification required the vehicle to drive at most 50km/h. However, a more realistic specification only demands that the speed of the vehicle is below 50km/h when close to a radar trap. Assume the trap is 60km away from the start position of the car. The extended specification merely requires two more output streams to compute the traveled distance and an adapted trigger condition. It uses the same technique showed in Equation 1.1, just for the position rather than the velocity, i.e.:

$$s(t) = \int_0^t \dot{s}(\tau)d\tau = \iint_0^t \ddot{s}(\tau)d^2\tau = \iint_0^t a(\tau)d^2\tau = \int_0^t \left(\int_0^\tau \ddot{s}(\iota)d\iota\right)d\tau = \sum_{i=0}^{t}\sum_{j=0}^{i-1}\Delta v_i$$

This results in the following specification where d_km is the number of kilometers the car has traveled so far:

```
input accel_mpss: Float64
output high_accel: Bool := abs(accel_mpss) > 4
trigger high_accel "Ride not smooth."

output Δv_mps: Float64 @1Hz := accel.aggregate(over: 1s, using: ∫)
output v_mps: Float64 @1Hz := v.offset(by: -1).defaults(to: 0.0) + Δv
output v_kmph: Float64 @1Hz := v_mps * 3.6
output d_km: Float64 @1Hz := d_km.offset(by: -1).defaults(to: 0.0) + Δd_m / 1000
trigger v_kmph > 50 ∧ d_km > 59.5 ∧ d_km < 60.1 "Speeding near radar trap."
```

The addition of new sensor values is similarly simple. Assume the vehicle is now equipped with a GPS module with alleged sensor frequency 10Hz. The specification checks the frequency and alerts the controller when the monitor receives too few samples. Moreover, it checks whether the computed position coincides with the measured

one up to a derivation threshold ϵ. This is a common pattern in a highly redundant, safety-critical system such as aircraft or autonomous cars. The same metric is computed in several different ways, implemented by different people, sometimes even in different programming languages, based on different sensors. If the resulting values match, there is a high confidence in the correctness of the value. If they diverge, a majority vote yields the most probable result and the sensor health estimate for the respective sensors deteriorates.

The extended specification featuring the cross validation looks as follows:

```
input accel_mpss: Float64
output high_accel: Bool := abs(accel_mpss) > 4
trigger high_accel "Ride not smooth."

output Δv_mps: Float64 @1Hz := accel.aggregate(over: 1s, using: ∫)
output v_mps: Float64 @1Hz := v.offset(by: -1).defaults(to: 0.0) + Δv
output v_kmph: Float64 @1Hz := v_mps * 3.6
output d_km: Float64 @1Hz := d_km.offset(by: -1).defaults(to: 0.0) + Δd_m / 1000
trigger v_kmph > 50 ∧ d_km > 59.5 ∧ d_km < 60.1 "Speeding near radar trap."

input gps: Float64
output gps_cnt: UInt64 @1Hz := gps.aggregate(over: 1s, using: count)
trigger gps_cnt < 10 "Few GPS samples."
output cross_diff: Float64 := abs(gps - d_km.hold().defaults(to: gps))
trigger cross_diff > ϵ "Estimated and measured position deviate."
```

Note that the GPS module in the simplified scenario only yields a single floating point number representing the x position of the car in km. The output stream `gps_cnt` counts the number of readings produced by the GPS module per second. If the count drops below 10, an alarm is raised because the module does not behave as expected. The count operation is another aggregation function that is predefined for sliding window expressions. Other prominent aggregation functions are extrema, summation, (co-)variance, and existential and universal quantification.

Lastly, the computation of `cross_diff` uses a *sample and hold* expression. This expression bridges the gap between periodic and event-based streams. When the accessing stream is evaluated, it takes the latest value of the accessed stream in a zero-order hold. This is different to the other accesses because they required a *compatible* timing between streams. Consider the expression in which `v_kmph` accesses `v_mps`. This is a *synchronous* access meaning that the evaluation of the accessing stream happens at the same point in time when the accessed stream is evaluated. Thus, the accessed value is "fresh", i.e., no zero-order hold is necessary.

Chapter 2

Related Work

Early work on formal runtime monitoring was mainly based on temporal logics [7, 8, 9, 2, 10, 11]. For many applications, these logics lacked expressiveness, giving rise to real-time temporal logics such as MTL [12] and STL [13]. For both logics, monitoring algorithms and implementations thereof exist [14, 15, 16, 17].

The main advantage of temporal logics is that monitors with formal guarantees on the space and time complexity can be synthesized for specifications.

These monitors can be abstract finite state automata, or concrete circuits: MBAC [18] translates PSL formulas into a monitoring automaton; FoCs [19] compiles sPSL [20] assertions to Verilog code, which can then be realized on programmable hardware such as Field-Programmable Gate Arrays (FPGA). Similarly, BusMOP [21] synthesizes hardware monitors out of past-time linear temporal logics. Jaksic et al. [22] introduced an algorithm to synthesize monitors for STL — and thus real-time languages — on an FPGA. Furthermore, the R2U2 tool [23, 24] introduced an FPGA implementation for real-time temporal specification based on MTL, featuring future-time specifications. All of these approaches, however, are restricted to monitoring specifications expressed in a logic. This allows for guarantees on the runtime behavior of the monitor. However, while these formal guarantees should not be neglected, a major drawback of logics is their expressiveness. Moreover, when the monitor is used for cyber-physical systems such as medical devices or autonomous vehicles, a simple yes or no verdict is often insufficient. Statistical information about the degree in which the execution was a failure helps the development process. Moreover, expressing arithmetic expressions with boolean properties is cumbersome unless the specification language provides appropriate primitives. For this reason, languages with more expressive native constructs such as arithmetic, and more expressive non-binary outputs have been studied.

One direction for increasing expressiveness is by monitoring first-order temporal logics. This can be achieved by using BDDs [25], Regular Expressions [26], rule-based systems [27, 5, 28], or SMT solvers [29]. Unfortunately, these approaches, while more expressive, still yield binary results only.

For this reason, temporal logics have been enriched by quantitative measures. These can be the edit distance to the desired result [30]. Similarly, the ratio of violation versus satisfaction is highly relevant. For this, the amount of satisfying and violating models [31, 32] or events [33, 34] can be counted. When considering real-time signals, the actual satisfaction and violation time on the real axis can be measured [35]. Mascle et al. [36] presented an approach for monitoring robust LTL [37, 38], a variant of LTL that naturally gives more detailed, non-binary verdicts. Lastly, aggregating expressions assessing the rate of satisfaction and dissatisfaction can be integrated into first-order temporal logics [39]. Moreover, when given a constant reference signal, signal integration gives a measure of deviation from the reference [40].

In the recent past, the collection of statistics became increasingly popular in monitoring tools [41, 42, 43]: A wide-spread, commercial, rule-based monitoring tool for networks, Snort [6], computes a pre-defined set of statistics efficiently. Rather than using a rule based language, Beep Beep 3 [44] uses a query language similar to SQL. Its connection to database systems allows for powerful aggregation functions and thus statistical measures. This comes at the hefty prive of requiring a heavy-weight database application on the system under scrutiny — which is impractical on many embedded devices. The Copilot [45] framework is a light-weight alternative based on synchronous languages [46, 47]. It transforms a declarative specification in a data-flow language into a simple C implementation. Due to the simplicity, guarantees on the space and time complexity of the monitor can be provided.

Also based on synchronous languages like Lustre [48, 49], and Esterel [50], stream-based monitoring approaches were studied in the form of the specification language Lola [51]. Lola forms the basis of the language formalized in this thesis. It is a descriptive language subsuming discrete temporal logics and can express properties concerning the past as well as the future. Apart from RTLola, there are two other languages based on Lola: TeSSLa and Striver.

TeSSLa [52] monitors piece-wise constant signals. Its output streams run on different timelines and can produce events at arbitrary points in time. This especially allows for Zeno[1] behavior that cannot be detected statically. This is impossible in RTLola since output streams are clocked based on input events which cannot be Zeno, or they are isochron, i.e., their evaluation behavior is determined statically. As a result, RTLola decouples input and output streams which allows for an efficient aggregation of values via sliding windows. This is not possible in TeSSLa. Moreover, as opposed to RTLola, TeSSLa relies on instrumented C code. As a result, it is not applicable on blackbox systems.

The other language that arose from Lola is Striver [53]. Striver also does not have a clean separation of input and output streams and lacks convenient and efficient native primitives for 0-order sampling and sliding window expressions.

[1] Zeno behavior, named after the Greek philosopher Zeno of Elea, requires a system to perform an infinite amount of operations in a finite amount of time.

An approach for compiling synchronous Lola has been presented by Maltry [54]. Later on, the work was extended to cover the entire language presented in this thesis [55].

The adequacy of Lola's and RTLola's expressiveness was demonstrated in case studies. The monitored systems were aircraft [56, 57, 55] and networks [58, 55]. In addition to that, there are first endeavors to apply the language on medical cyber-physical devices[59]. This thesis presents the underlying case studies.

Chapter 3

Preliminaries

In this chapter, we will lay the basis for the remainder of the thesis by introducing some notation and concepts as well as the language on which RTLola is based.

3.1. Notation

First, we introduce is a notation to transform a binary value into a corresponding integer value.

Definition 1 (Indicator)

The *(boolean) indicator* $\mathbb{1}_{\phi}$ yields 1 if ϕ is true and 0 otherwise. **Def.** Indicator

$$\mathbb{1}_{\phi} := \begin{cases} 1 & \text{if } \phi \\ 0 & \text{otherwise} \end{cases}$$

The next definition concerns a short-hand notation for a series of similar elements. It allows for a concise notation without the need of resorting to the abbreviating ellipsis (…) notation.

Definition 2 (Families)

A *family* is a sequence of similar elements. It consists of an enumerable set S with order $<_S$, and a unary function $f\colon S \to S'$. The order of S induces the order of the resulting sequence, and f produces the respective elements. If unambiguously possible, f is left implicit, so $i \in \mathbb{N}$ represents the identity function on $\mathbb{N}$. **Def.** Family

$$(f(a))_{a \in S} := f(a_1), f(a_2), \ldots \qquad \text{given } \bigcup_{i \in \mathbb{N}} \{a_i\} = S \wedge \forall i \in \mathbb{N}\colon a_i <_S a_{i+1}$$

3.2. Basic Definitions

In this section, we will recall the definitions of some basic concepts concerning lattices and define a notion of interval partitioning.

Definition 3 (Partial Order)

Def. Partial Order

A *partial order* on S is a binary relation $\leq \subseteq S \times S$ that satisfies three criteria:

Reflexivity $\forall a \in S : a \leq a$

Anti-Symmetry $\forall a, b \in S : a \leq b \wedge b \leq a \implies a = b$

Transitivity $\forall a, b, c \in S : a \leq b \wedge b \leq c \implies a \leq c$

Note that an order defined with a strict less $<$ can be a partial order if equality $\approx$ is defined based on non-relation of elements, i.e. $a \not< b \wedge b \not< a \implies a \approx b$ for any a and b.

The concept of partial orders is integral for lattices. Note that there are two commonly used definitions for lattices, a set-theoretic and an algebraic one. Both are equivalent, we will use the latter one.

Definition 4 (Meet-Semilattice)

Def. Meet-Semilattice

A *i*s a set S with partial order $\sqsubseteq$ and binary meet operator $\sqcap$. The meet operation needs to be defined on any two elements in S and satisfy the following criteria:

Closedness $\forall a, b \in S : a \sqcap b \in S$

Associativity $\forall a, b, c \in S : (a \sqcap b) \sqcap c = a \sqcap (b \sqcap c)$

Commutativity $\forall a, b \in S : a \sqcap b = b \sqcap a$

Idempotency $\forall a \in S : a \sqcap a = a$

Def. Bounded Meet-Semilattice

The $\sqcap$ operator on two elements $a, b \in S$ then results in the greatest lower bound of a and b in S w.r.t. $\sqsubseteq$. The semilattice is *bounded* if S contains an identity element ϵ with $\epsilon \sqcap a = a$ for any $a \in S$.

Related to the closedness property of the lattice, recall the definition of a transitive closure of a binary relation.

Definition 5 (Transitive Closure)

Def. Transitive Closure

The *transitive closure* of a binary relation $\mathcal{R} \subseteq S \times S$ is a set $\mathcal{R}' \supseteq \mathcal{R}$ satisfying:

$$\forall a, b, c \in S : a\mathcal{R}b \wedge b\mathcal{R}c \implies a\mathcal{R}'c$$

It can always be constructed by successively adding missing elements to $\mathcal{R}'$ until the property is satisfied.

Lastly, we define a special partition for intervals. It essentially splits an interval into a sequence of different non-overlapping, consecutive chunks. The sequence of chunks preserves the order in the original interval.

Definition 6 (Ordered (Interval-) Partition)
Let $\mathcal{I} = (I_i)_{i \leq k}$ for some k. $\mathcal{I}$ is an *ordered partition* of $\mathcal{X} \subseteq \mathbb{N}$ iff

Def. Ordered Partition

$$\bigcup_{i \leq k} I_i = \mathcal{X} \wedge \forall i \leq k \colon \emptyset \neq I_i = [\ell, u] \wedge \emptyset \neq I_{i+1} = [\ell', u] \implies \ell' = u + 1$$

The first criterion ensures that $\mathcal{I}$ covers $\mathcal{X}$ entirely, the second ensures that the subsequences are non-overlapping and ordered.

Example 3.2.1. $([2^i, \ldots, 2^{i+1} - 1])_{i \in \mathbb{N}}$ is an ordered interval partition of the natural numbers. △

3.3. Lola

Lola [51] is a strongly-typed stream-based specification language. Any specification consists of input streams, output streams and triggers. Input stream declarations only contain type information about the expected input data, whereas output streams also declare a stream expression. This expression may contain conditional expressions, constants, arbitrary k-ary operators, and lookup expressions with discrete offsets. As opposed to RTLola, these offsets can be positive, in which case they refer to a future value of a stream. The evaluation of the expression is delayed until the referenced value is available.

As a consequence of this, Lola allows for unbounded future references, These occur if a stream s refers to the next value of s' and s' refers to the next value of s. In this case, the evaluation is only possible as soon as the end of the input trace is reached. If the length of the trace is statically unknown, the memory consumption of the monitor is equally unbounded. Such specifications are considered not *efficiently monitorable*.

However, this problem can be determined statically based on the dependency graph (DG) of a specification. The DG consists of nodes representing streams and edges representing stream accesses where the offsets are the weights of the edge. If there are no positive cycles in the DG, every stream only depends on values that will be available after a bounded amount of steps.

Theorem 1 (Efficiently Monitoriable [51])**.** *A specification is efficiently monitorable if there is no cycle with positive weight in the dependency graph.*

The dependency graph helps to compensate for another problem of Lola: Not every Lola specification has exactly one solution for the evaluation of stream expressions — some have multiple, others none. Image an output stream s which expression consists of an access to s itself with an offset of 0. As a result, s needs to assume the value

of s, a condition that holds vacuously. Thus, any evaluation of the expressions is valid. Similarly, consider a boolean stream, i.e. its expression can only yield true or false. If this expression is a logical negation of a lookup targeting itself with an offset of 0, then no evaluation result is valid for this stream. Fortunately, there is a sufficient criterion on the DG that identifies such constellations.

Theorem 2 (Unique Evaluation Model [51])**.** *A specification has a unique evaluation model if the dependency graph has no 0-weight cycle. The inverse direction does not hold.*

Lastly, note that there is a monitoring algorithm for any efficiently monitorable Lola specification with a unique evaluation model. The algorithm requires constant memory in the length of the trace.

Chapter 4

An Understandable Specification Language

The stream-based specification language RTLola is designed to satisfy two seemingly contradictory goals: simplicity and expressiveness. Simplicity is a subjective metric and is influenced by the conciseness and readability of a specification. In general, abstract concepts are less readable and more expressive than concrete ones. As an example, consider the highly abstract `flatmap` function found in many higher-order programming languages like Haskell and Scala. It is polymorphic and of type:

```
flatmap<T,A>: List<T> -> (T -> Option<A>) -> List<A>
```

Semantically, it takes every element of a list and applies a function that may or may not return a value, indicated by the monadic `Option` type. It returns a list of the results of the function application granted it was not `None`. Compare this to the less abstract and less expressive, polymorphic `filter` function with type:

```
filter<T>: List<T> -> (T -> Bool) -> List<T>
```

It takes a list as an argument and retains all values for which the second argument, a boolean classifier, returns a positive result.

Technically, `filter` is a superfluous function as it can be expressed with the `flatmap` function:

```
filter l p = flatmap l (λ x: if p x then Some(x) else None)
```

Yet, it is part of most — if not all — standard libraries along with `flatmap`. The reason is the relative readability of `filter` when compared with `flatmap`. The name `filter` already indicates what the result will be, i.e., a shorter version of the list where each element satisfies the filter criterion. The name `flatmap`, however, only conveys the abstract meaning of the function but not about the resulting list, which would require information about the second argument. Thus, developers tend to use the less expressive, more readable `filter` function whenever possible.

This principle was a leading factor when designing RTLola. Expressiveness is without a doubt crucial for a specification language. An overly clumsy and cumbersome language has slim chances of ever being widely adopted[1]. Moreover, an illegible specification can contain significant errors that remain hidden on first glance. For a safety-critical component such as a runtime monitor, hidden errors entail potentially dangerous consequences and should thus be ruled out as much as possible. So, the design of RTLola is focused on providing a language that is easy to understand even for novices. It still provides enough functionality such that even complicated properties can be specified easily.

This section presents the "vanilla" syntax of RTLola, i.e., the basic building blocks of a specification without syntactic sugar. It exhibits the entire expressiveness of RTLola while easing the process of defining the formal semantics and understanding the evaluation algorithm. Real specifications can then use syntactic sugar and predefined functions as defined in Section 6.1 to circumvent clumsy syntactic structures. A *desugarizing* step replaces syntactic sugar by the respective, equivalent vanilla RTLola constructs.

The concrete syntax is transformed into the *abstract syntax tree*. Based on this, we will define static analyses: a dependency and a type analysis. Both target the detection of inconsistencies in the specification before deployment of the monitor.

4.1. Concrete Syntax

An RTLola specification declares *input streams* that constitute the entry point of data. Further, it defines *output streams* that transform, aggregate, and analyze the input data to produce more refined output data. This data can either be logged or be target of a *trigger*, which raises an alarm if a stream produces a positive boolean result.

The abstract representation for a specification is the *abstract syntax tree*, which has $n^{\downarrow} + n^{\uparrow} + n^{!}$ children, representing all input streams, output streams and triggers, respectively. With slight abuse of notation, let AST be a function transforming a concrete specification into its respective AST.

Note that in the remainder of this thesis, we will indicate the kind of streams using a marker as superscript. A down arrow such as in $s^{\downarrow}$ symbolizes inputs, an up arrow ($s^{\uparrow}$) symbolizes outputs, and a bang ($s^{!}$) symbolizes triggers. Lastly, a vertical line (s^{-}) indicates that the subject is either an input, or an output.

The i^{th} input stream is of form:

```
input a: T
```

The respective AST is $s_i^{\downarrow} = (a, AST(T))$. The name of the stream is a one-to-one copy and irrelevant for the semantics. The type T is transformed into an AST object. For conciseness of notation, we define $s_i^{\downarrow}.name := a$ and let $T_i^{\downarrow}$ be $AST(T)$.

The syntax for output streams is similar, yet features two more components: an expression `e` and an evaluation frequency `nHz`. Let the following be the j^{th} output stream.

```
output a: T @nHz := e
```

[1] Yet, some recent, wide-spread languages like Javascript beg to differ.

The respective AST is $s^{\uparrow}_j = (a, AST(T), n, AST(e))$ for a natural number n. For brevity, let $s^{\uparrow}_j.name = a, s^{\uparrow}_j.ext = n, s^{\uparrow}_j.expr = AST(e)$, and $T^{\uparrow}_j = AST(T)$.

Remark 4.1.1 (A Note on Time and its Units)**.** *We require n to be a natural number and enforce the unit hertz for it. This is unnecessarily restrictive, the algorithm presented in this thesis is theoretically capable of working with any kind of frequency. The actual order of magnitude induced by the unit, i.e., hertz, millihertz, etc., is irrelevant, yet introduces an additional complication into the semantics. Thus, for simplicity, we assume hertz. With this in mind, one can easily see that the restriction to a natural number can be replaced by a restriction to positive rational numbers. The remaining irrational numbers are utterly irrelevant in practice because a computer cannot wait for exactly $\frac{1}{\pi}$ seconds. From a theoretical standpoint, the presented approach is still applicable to specification with irrational frequencies. However, it makes treatment of frequencies at some points unnecessarily cumbersome. For these two reasons, we disallow them statically in this chapter.*

The specified frequency is part of the stream's type and optional, so the following syntax for $s^{\uparrow}_j$ would yield the AST $(a, AST(T), \bot, AST(e))$.

```
output a: T := e
```

Lastly, a trigger consists of a stream name and a message. In the AST, the stream name is replaced by a reference to the stream with the specified name or $\bot$ if no such stream exists.

```
trigger a "msg"
```

$$s^{!}_k = (s, msg) = \begin{cases} (s^{\uparrow}_j, msg) & \text{if } s^{\uparrow}_j.name = a \\ (\bot, msg) & \text{otherwise} \end{cases}$$

We again abbreviate $s^{!}_i.tar = s$ and $s^{!}_i.msg = msg$.

4.1.1. Expressions

Expressions in RTLola consist of the usual operations such as constants, arithmetic operations, and conditionals. All of these constructs are collected in arbitrary function expressions: constants are 0-ary functions, n-ary arithmetic operations become n-ary functions and conditionals become ternary functions. Infix operations such as `a + b` and the common `if b then e1 else e2` notation are syntactic sugar for `plus(a, b)` and `ite(b, e1, e2)`, respectively. Note that the conditional can only be encoded as a function because the semantics of RTLola expressions is free of side-effects. Further discussion follows in Remark 6.1.1.

→ Sec. 6.1, Page 70

In addition to these operations, RTLola expressions contain *lookups* that allow output streams to access other streams in one of three ways, see Table 4.1.

Synchronous Lookups A stream accesses another stream *synchronously*, i.e. whenever the accessee stream produces a value, the accessor stream gets access to it. This reflects a push-based paradigma where data is pushed from accessee to accessor.

Syntax	Name
`e.offset(by: -n)`	Offset
`e.defaults(to: e)`	Default
`f(e1,...,en)`	Function
`s`	Synchronous Lookup
`s.hold()`	Sample&Hold Lookup
`s.aggregate(over: δs, using: γ)`	Sliding Window

Table 4.1.: Concrete syntax of RTLola expressions and their corresponding names.

Sample & Hold A stream accesses another stream *asynchronously*. The accessor stream gets access to the latest value computed for the accessee stream. The timing is dictated by the accessor, so this offset reflects a pull-based paradigma where the accessor pulls data from the

Sliding Window A stream can access all values of another stream that occurred in a certain real time period. This is called a sliding window because the window of relevant values slides continuously over the real time axis. The sequence of values is aggregated with an aggregation function. The duration of the window is a natural number of seconds. This limitation can be resolved as explained in Remark 4.1.1.

Some of these lookups can fail, resulting in an *optional* value. The default operation remedies this by providing a default value in case the lookup failed.

The output of the *AST* function is then:

$$AST(\texttt{s.offset(by: -n)}) := \begin{cases} Offset(s_i^-, n) & \text{if } s_i^-.name = s \\ Offset(\bot, n) & \text{otherwise} \end{cases}$$

$$AST(\texttt{e.defaults(to: d)}) := Default(AST(d), AST(e'))$$

$$AST(\texttt{f(e1,...,en)}) := Func(f, AST(e1), \ldots, AST(en))$$

$$AST(\texttt{s}) := \begin{cases} Sync(s_i^-) & \text{if } s_i^-.name = s \\ Sync(\bot) & \text{otherwise} \end{cases}$$

$$AST(\texttt{s.hold()}) := \begin{cases} Hold(s_i^-) & \text{if } s_i^-.name = s \\ Hold(\bot) & \text{otherwise} \end{cases}$$

```
input a: Int8
input b: Bool
input c: Float32
output d: Int8 @5Hz := a.aggregate(over: 12s, using: Σ)
output e: Float32 := multiply(
    a.offset(by: -3).defaults(to: 99),
    f.hold().defaults(to: 1)
  )
output f: Float32 := gt(ite(b, e, add(c, 4.0)), 9000.0)
trigger f "It's over 9000.0!"
```

Figure 4.2.: A "vanilla" RTLola specification showcasing the concrete syntax elements. `multiply` and `add` are binary multiplication and addition, `ite` is the ternary conditional operator. 0-ary functions representing constants are in-lined.

$$AST(\texttt{s.aggregate(over:}\delta\texttt{s, using:}\gamma\texttt{)}) \coloneqq \begin{cases} Window(s_i^-, \delta, \gamma) & \text{if } s_i^-.name = s \\ Window(\bot, \delta, \gamma) & \text{otherwise} \end{cases}$$

Note that — similar to streams — we assign unique indices to sliding window expressions. The i^{th} window occurring in the specification is thus called w_i.

The BNF for the concrete syntax can be found in the appendix, Section A.1

→ Sec. A.1, Page 85

The validity of an RTLola specification is based on several factors. The first one is the syntactic validity, which is defined as follows.

Definition 7 (Syntactic Validity)

An RTLola specification is *syntactically valid* iff

Def. Syntactic Validity

- all stream and trigger definitions conform to the concrete syntax stated above,
- $AST(s)$ can be computed without violating a condition. This especially includes that all stream names can be resolved.
- all names of streams are unique, i.e., $\forall i, j : s_i^-.name = s_j^-.name \implies i = j$

Example 4.1.1. Figure 4.2 shows a specification written in "vanilla" RTLola, i.e., without any syntactic sugar apart from in-lined constants. The first three lines declare input streams of type integer with bit-width 8, a boolean stream, and a floating point stream with bit-width 32. Output `d` declares a sliding window over stream `a`, integrating all values of the last 12s. Output stream `e` multiplies the fourth-to-last value of `a` defaulting to 99, and the last value of `c` defaulting to 1. Note that the offset is −3 and results in the fourth-to-last value because the "first-to-last" value is the last value and represents an offset of 0. Lastly, `f` compares either `d` or `c + 4.0` with 9000. If it is greater, the trigger goes off and alarms the user or controller thusly.

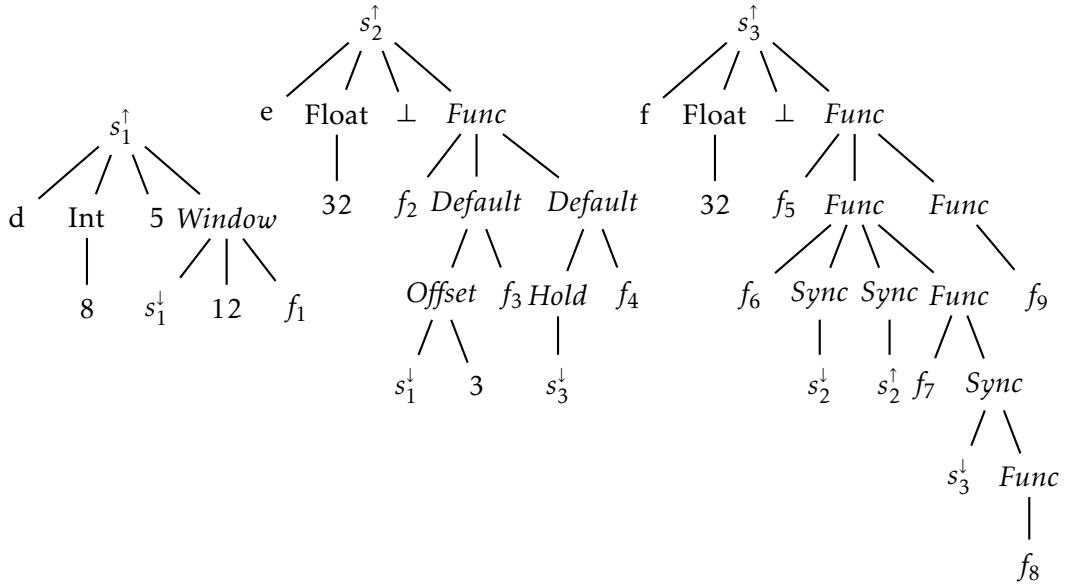


Figure 4.3.: Abstract syntax tree for the output streams of the RTLola specification in Figure 4.2.

The AST consists of seven root level nodes, three representing the inputs, one for the trigger, and three representing the outputs. The latter are best illustrated as trees, see Figure 4.3, whereas the input streams, trigger and functions look as follows:

$$s_1^\downarrow = (a, Int(8)) \qquad f_1((x_i)_{i\le k}) = \sum_{i=1}^{k} x_i \qquad f_6(x,y,z) = \begin{cases} y & \text{if } x \\ z & \text{otherwise} \end{cases}$$

$$s_2^\downarrow = (b, Bool) \qquad f_2(x,y) = x \cdot y$$

$$s_3^\downarrow = (c, Float(32)) \qquad f_3 = 99 \qquad f_7(x,y) = x + y$$

$$s_1^! = (s_3^\uparrow, \text{“It's over 9000.0!”}) \qquad f_4 = 1 \qquad f_8 = 4$$

$$f_5(x,y) = x > y \qquad f_9 = 9000$$

△

From now on, we will always refer to the AST of a specification rather than its concrete syntactic form. Moreover, $Stream = Stream^\downarrow \dot\cup Stream^\uparrow \dot\cup Stream^!$ denotes the set of streams comprised of the set of input streams $Stream^\downarrow = \{s_i^\downarrow \mid i \le n^\downarrow\}$, output streams $Stream^\uparrow = \{s_i^\uparrow \mid i \le n^\uparrow\}$, and triggers $Stream^! = \{s_i^! \mid i \le n^!\}$, respectively. Lastly, let $\mathcal{W} = \{w_i \mid i \le n^w\}$ be the set of all window expressions.

4.2. Dependency Graph

While the abstract syntax tree captures the syntax of an RTLola specification, the dependency graph is concerned with semantic dependencies between streams. Intuitively, an output stream s depends on another stream s' if the evaluation of the stream expression of s requires access to stream values of s'. The dependency graph captures these dependencies and is required for semantic checks such as the type analysis and the existence of an evaluation model. Moreover, it allows us to determine a static upper bound on the memory consumption of a monitor for the specification.

Definition 8

The expression of an output stream contains the information which streams are accessed, i.e. on which streams it potentially depends. A dependency is a triple consisting of the accessing stream, a weight capturing the temporal properties of the access, and the target stream. The computation is recursive over the structure of the stream expression of $s_i^{\uparrow}$.

$$dep_{s_i^{\uparrow}}(Offset(s_j^{-}, n)) := \{(s_i^{\uparrow}, n, s_j^{-})\}$$

$$dep_{s_i^{\uparrow}}(Default(e, e')) := dep(e) \cup dep(e')$$

$$dep_{s_i^{\uparrow}}(Func(f, a_1, \ldots, a_n)) := \bigcup_{0<i\leq n} dep(a_i)$$

$$dep_{s_i^{\uparrow}}(Sync(s_j^{-})) := \{(s_i^{\uparrow}, 0, s_i^{-})\}$$

$$dep_{s_i^{\uparrow}}(Hold(s_j^{-})) := \{(s_i^{\uparrow}, 0, s_j^{-})\}$$

$$dep_{s_i^{\uparrow}}(Window(s_i^{-}, \delta, \gamma)) := \{(s_i^{\uparrow}, (\delta, \gamma), s_j^{-})\}$$

This allows us to define the dependency graph.

Definition 9 (Dependency Graph)

The *dependency graph* of a specification is a directed, multi-graph $DG = (V, E)$ with weighted edges. Its vertices are streams and the edges reflect dependencies between streams:

Def. Dependency Graph

$$V := Stream$$

$$E := \bigcup_{1\leq i\leq n^{\uparrow}} dep_{s_i^{\uparrow}}(s_i^{\uparrow}.expr) \cup \bigcup_{1\leq i\leq n^{!}} \{(s_i^{!}, 0, s_i^{!}.tar)\}$$

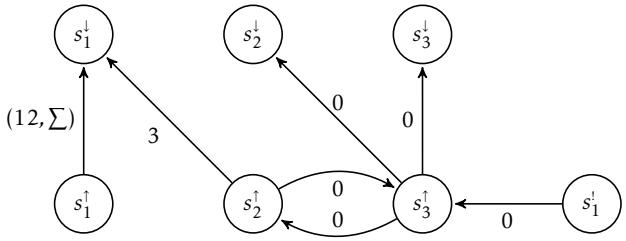


Figure 4.4.: The dependency graph for the specification in Example 4.1.1.

Example 4.2.1. Recall the specification from Example 4.1.1. Figure 4.4 shows the respective dependency graph. As for all specifications, input streams are sink nodes whereas triggers are source nodes. However, output stream $s_1^{\uparrow}$ can also be a sink node. Nodes like these do not serve any purpose in the specification and can thus be purged prior to evaluation. However, it might be beneficial to log all values of input and output streams during execution for post-deployment analysis. In this case, sink nodes cannot be purged.

△

4.2.1. Evaluation Models

Def. Evaluation Model

The dependency graph grants some insights into the requirements on the expression evaluation. Without detailing the semantics of RTLola specifications we can already identify potential problems. Intuitively, an *evaluation model* of a specification is a set of infinite sequences of values where each sequence represents a single stream. The values of the sequence need to be in the domain of the stream. As an example consider an input stream $s^{\downarrow} = (a, Int(8))$. A valid evaluation model of this stream is $(x \mod 128)_{x \in \mathbb{N}}$.

While an input stream imposes very little constraints on its model, the expression of output streams dictates the model mostly explicitly. The following specification uniquely defines its model as $(x \mod 128)_{x \in \mathbb{N}}$:

```
output b: Int8 := b.offset(by: -1).defaults(to: 0)
```

However, this is not always the case. Consider the following specification:

```
output x: Int8 := y
output y: Int8 := x
```

Intuitively, the stream `x` copies the value of `y` and `y` copies the value of `x`. As a result, there is more than one valid model for these streams; namely any two identical sequences over natural numbers in the domain of 8 bit integers.

In contrast to that, the following specification does not have a single evaluation model because it constitutes a logical contradiction.

```
output x: Bool := negation(y)
output y: Bool := x
```

Def. Well-Definedness

A specification is said to be *well-defined* if a unique model exists. This problem al-

ready arose for Lola specifications [47]. Here, D'Angelo et al. proved that a unique model exists if the dependency graph of a specification does not contain a cycle with weight 0, cf. Theorem 2. We will re-instate this result for RTLola specifications in Proposition 12. Here, we show that the absence of 0-weight cycles implies the existence of an *evaluation order* (formally defined later), which implies well-definedness. The intuition behind the proof is simple: the difference between sample and hold operations and synchronous lookups is only concerned with timing. Since this is irrelevant for the number of valid models, both operations behave equivalently. Window operations impose restrictions on the evaluation order of streams but behave as constants afterwards. Thus, the result for Lola is applicable to RTLola specifications.

→ Sec. 3.3, Page 14
→ Sec. 4.4, Page 45

Remark 4.2.1 (Sliding Windows as 0-Dependency)**.** *For the intends and purposes of the remainder of this section, sliding window edges are considered 0-weight edges unless stated otherwise. This is because the evaluation of a sliding window requires its target stream to be evaluated the same way a synchronous lookup does.*

The absence of 0-weight cycles is captured in a the *well-formedness* property.

Definition 10 (Well-formedness)

An RTLola specification is *well-formed* if there is no cycle with weight 0 in the dependency graph.

Def. Well-formed

Example 4.2.2. The specification from Example 4.1.1 is not well-formed. This becomes evident in the dependency graph in Figure 4.4 due to the 0-cycle between $s_2^{\uparrow}$ and $s_3^{\uparrow}$. △

4.2.2. Evaluation Order

The evaluation of an RTLola specification requires to evaluate the expressions of output streams. In this process, the order in which expressions are evaluated influences the outcome, as can be seen in the following example.

Example 4.2.3. Consider the following specification:

```
input i: Int8
output a: Int8 = b.offset(by: -1).defaults(to: i)
output b: Int8 = multiply(2, i)
```

When evaluating this specification, the input stream `i` is synchronously accessed by both output streams, so it needs to be evaluated first. Stream `a` accesses `b` with a negative offset, meaning it accesses the *second to last* value. This, however, assumes that `b` was already evaluated. Inspecting the stream expression of `b` reveals that it only depends on `i`. Thus, it can be evaluated before `a`, resulting in a correct evaluation.

Therefore, the appropriate evaluation order $\prec$ for this specification is $\texttt{i} \prec \texttt{b} \prec \texttt{a}$. △

Yet, there are specifications where an evaluation order does not solve all dependencies properly.

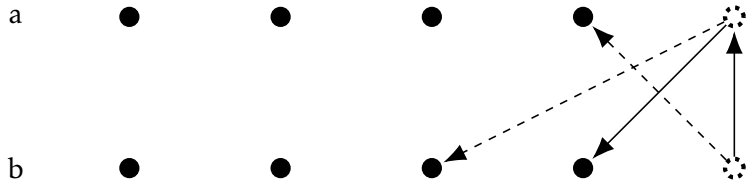


Figure 4.5.: A timing diagram showing that an offset shift is required for the evaluation. Solid arrows represent intended lookup targets, dashed arrows indicate the lookup result when resolving offsets naïvely.

Example 4.2.4. Consider a slightly modified version of the specification from the former example.

```
input i: Int8
output a: Int8 = b.offset(by: -1).defaults(to: i)
output b: Int8 = a.hold().defaults(to: multiply(2, i))
```

In this case, yet again, `i` needs to be evaluated first. The dependency between `a` as accessor of `b` persists and is complemented by `b` accessing `a` synchronously. Note that this specification is well-formed since one of the resulting edges has weight -1 whereas the other has weight 0. In this scenario, evaluating `b` first, accessing `a`'s latest value is incorrect since `a` needs to produce a new value first. Evaluating `a` first, however, results in an access to `b`'s second to latest value, which is incorrect because `b` was not evaluated, yet. The value that ought to be accesses in this case is the *latest* value of `b` *before* evaluating `b`. Figure 4.5 illustrates the dependency structure of the streams.

△

Pseudo Evaluation

This problem is solved by initiating a *pseudo evaluation phase* in which all streams that will be evaluated get a new pseudo-value. This value is replaced as soon as the actual value is computed. During the evaluation of expressions, the specified offsets can be resolved regularly. These accesses are correct assuming that the evaluation order is obeyed.

The computation of the evaluation order is based on the dependency graph and follows the intuition gained from the former two examples: When a stream s accesses another stream s', then s' needs to be less than s according to the order. Thanks to the pseudo extension phase, accesses with a non-zero offset can be disregarded for the evaluation order. The access refers to the a past value, which consequently is already accessible and does not impose a restriction on the evaluation order.

Definition 11 (Evaluation Order)

Def. Evaluation Order

The *evaluation order* $\prec$ is a partial order on streams. Incomparable streams can be evaluated in an arbitrary order because they either do not depend on each other, or the dependencies refer to accessible values. The order reflects the structure of the dependency graph $DG = (V, E)$. The evaluation order is the transitive closure of a relation satisfying the following rules:

1. $\forall i, j \colon s_i^{\downarrow} \prec s_j^{\uparrow}$
2. $s_i^{\uparrow} \prec s_k^{\uparrow} \wedge s_k^{\uparrow} \prec s_j^{\uparrow} \implies s_i^{\uparrow} \prec s_j^{\uparrow}$
3. $(s_i^{\uparrow}, x, s_j^{-}) \in E \wedge (x = 0 \vee x = (\delta, \gamma)) \wedge s_i^{\uparrow} \neq s_j^{-} \implies s_j^{-} \prec s_i^{\uparrow}$

The first rule ensures that input streams are evaluated first. While not strictly necessary, it eases the evaluation process. The second rule ensures the transitivity of $\prec$. Lastly, the third rules only concerns 0-weight and window dependencies because everything else is resolved with the pseudo extension. It especially includes prohibiting that an access of a stream on itself influences the order.

Note that triggers are not part of the evaluation order. They always access a single other stream and cannot be accessed themselves. Thus, they can always be evaluated in a last step.

Proposition 3 (Existence of Evaluation Order)**.** *Every well-formed specification has an evaluation order.*

Proof. We first construct a relation $\prec \subseteq \mathit{Stream} \times \mathit{Stream}$ that satisfies all criteria by construction. We then show that it must be a partial order. The first rule is satisfied when declaring all input streams less than all output streams.

$$\prec^1 := \bigcup_{s_i^{\downarrow} \in \mathit{Stream}^{\downarrow}} \bigcup_{s_j^{\uparrow} \in \mathit{Stream}^{\uparrow}} \{(s_i^{\downarrow}, s_j^{\uparrow})\}$$

Next, $\prec^2$ satisfies the first and third rule by construction.

$$\prec^2 := \prec^1 \cup \bigcup_{\substack{(s_i^{\uparrow}, x, s_j^{-}) \in E \\ x = 0 \vee x = (\delta, \gamma) \\ s_i^{\uparrow} \neq s_j^{-}}} \{(s_j^{-}, s_i^{\downarrow})\}$$

Lastly, $\prec^3$ is the transitive closure of $\prec^2$.

The three criteria for a partial order are transitivity, given by the transitive closure, reflexivity, and anti-symmetry. Reflexivity for output streams follows from the fact that reflexive relations are explicitly excluded in the construction and the third rule for evaluation orders. Reflexivity for input streams does not immediately follow from the first rule for evaluation orders. However, the construction only adds relations between inputs and outputs as well as between different outputs.

Anti-symmetry follows from the absence of 0-weight cycles. For this, assume the relation was not anti-symmetric. Yet again, the construction does not allow for a relation between two input streams. Thus:

$$\exists i, j \colon s_i^{\uparrow} \prec^3 s_j^{\uparrow} \wedge s_j^{\uparrow} \prec^3 s_i^{\uparrow}$$

We claim for any pair of output streams $s_i^{\uparrow}$ and $s_j^{\uparrow}$:

$$\forall i, j \colon s_i^{\uparrow} \prec s_j^{\uparrow}$$
$$\iff \exists s_{k_1}^{\uparrow} \dots s_{k_n}^{\uparrow} : (s_i^{\uparrow}, x_1, s_{k_1}^{\uparrow}), (s_{k_1}^{\uparrow}, x_2, s_{k_2}^{\uparrow}), \dots, (s_{k_{n-1}}^{\uparrow}, x_n, s_{k_n}^{\uparrow}), (s_{k_n}^{\uparrow}, x_{n+1}, s_j^{\uparrow}) \in E$$

Here, each x_i is either 0 or a (δ, γ).

If the relation between $s_i^{\uparrow}$ and $s_j^{\uparrow}$ arose from the third rule, the claim follows by construction. Otherwise, since the first rule is not applicable on two output streams, the relation is the result of the transitivity rule. Thus, there is an intermediate stream $s_k^{\uparrow}$ with $s_i^{\uparrow} \prec s_k^{\uparrow} \prec s_j^{\uparrow}$. Now, the argument is applicable recursively. Either both $\prec$ relations arose due to the third rule, leading to the desired 0-path through DG, or we resolve another transitivity rule the same way. The argument is well-founded because any such path is either acyclic and finite because V is finite, or cyclic, contradicting the absence of 0-cycles in DG.

Using this information, we can conclude that $s_i^{\uparrow} \prec s_j^{\uparrow} \wedge s_j^{\uparrow} \prec s_i^{\uparrow}$ constitutes a 0-cycle, rendering it a contradiction to the assumption.

Thus, the constructed relation is a partial order and an evaluation order. □

The semantics heavily relies on the evaluation order. For this, however, a different, equivalent representation is more convenient.

Definition 12 (Evaluation Layer)

Def. Evaluation Layer

The *evaluation layer* is an equivalent representation of $\preceq$. If $Layer(s_i^{-}) = k$ then there is a strictly decreasing sequence of k streams w.r.t. $\prec$ starting in s_i^{-}. Intuitively, a stream from layer k only depends on streams from layers $k-1$ or lower.

$$Layer(s_i^{-}) := 1 + \max\{Layer(s_j^{-}) \mid s_j^{-} \prec s_i^{-}\}$$

In the following, λ^* denotes the maximum layer, i.e.,

$$\lambda^* := \max\{\lambda \mid \exists s_j^{-} \colon \lambda = Layer(s_j^{-})\}$$

4.3. Type System

Value Type

RTLola is a strongly typed language. There is two notions of types. The *value type* provides information about the domain and shape of a single value. RTLola's value types also occur in many other strongly typed programming languages such as Java or Rust. All value types except *Bool* are annotated with a number that indicates how many bits are used to represent a single value of this type. While the value type indicates the *spatial* behavior of a value, the second dimension of types in RTLola argues about its

Pacing Type

temporal behavior. The *pacing type* indicates when new values arrive or get computed. To understand the pacing type, it is easier to think of expressions rather than singular

values. If an expression has the value type $Int(8)$ and pacing type 3Hz, the expression is evaluated thrice per second and each resulting value is of type $Int(8)$, i.e. a signed integer representable with 8 bits. The pacing type either states that a value is produced periodically e.g. with 3Hz, or sporadically.

A type is a tuple of a value and a pacing type; every stream and every expression in RTLola is required to have such a type. The concrete syntax of RTLola requires that every stream is annotated with its value type and may be annotated with its pacing type when its periodic. This is not always necessary as most types can be inferred. Consider, for example, a specification with an input stream s of type $Int(8)$ and an output stream that only accesses s. The value type of the output is thus the same as the one of s. Regardless, an annotation can change the type: if the output is declared as $Int(16)$, the value of type $Int(8)$ can be *coerced* into 16 bits by repeating the most-significant bit 8 times.

This section introduces RTLola's type system and defines type validity. For type inference and type omission, refer to Section 6.1.1.

→ Sec. 6.1, Page 72

We will first define types.

Definition 13 (Value Types)

In RTLola, a single value is of one of the following types:

Def. Value Type

$$VT := \{Bool, Int(x), UInt(x), Float(y) \mid x \in \{8, 16, 32, 64\}, y \in \{32, 64\}\}$$

These types represent boolean values, signed integers, unsigned integers, and floating point numbers, respectively.

The type annotations after a colon in the specification is always a value type. The optional value after the at sign ("@") indicates the pacing type.

Definition 14 (Pacing Types)

Def. Pacing Types

Pacing types describe the temporal behavior of streams and expressions, i.e., when new values become available. This can either be periodically, i.e., after pre-defined time intervals, or sporadically, i.e., at unknown points in time.

In the former case, the pacing type is a *periodic type* $\pi \in PT$. If an output stream has periodic type $\pi = 3$, it is evaluated thrice per second.

Def. Periodic Type

A sporadic pace is captured in an *event type* ET. If an output stream s has event type $\iota = \{a, b\}$ for input streams a and b, s is evaluated whenever new values for a *and* b arrive. Since the arrival of input values is uncontrollable and thus sporadic in the worst case, the evaluation of s is also considered sporadic. If another output stream only accesses s, it has the same event type as s. The event type thus represents immediate or transitive dependencies of output streams to input streams.

Def. Event Type

Formally, an event type is a subset of input streams. The periodic type is a natural number that divides the least common multiple (lcm) of all periodic types declared in the specification. Recall Remark 4.1.1 for why natural numbers are not a restriction here. Moreover, the greatest common multiple of all occurring periods needs to divide the greatest common divisor (gcd).

→ Sec. 4.1, Page 17

Let $P^{spec} := \bigcup_{s_i^{\uparrow} \in Stream^{\uparrow}} \{s_i^{\uparrow}.ext\}$.

$$ET := 2^{Stream^{\downarrow}} \tag{4.1}$$

$$PT := \{p \mid p \in \mathbb{N} \wedge \gcd(P^{spec}) \,\big|\, p \wedge p \,\big|\, \mathrm{lcm}(P^{spec})\} \tag{4.2}$$

Note that input streams cannot have a periodic type. This is because the monitor has no control over input streams and thus cannot enforce the arrival of new values. Thus, every input stream $s_i^{\downarrow}$ has type $\{s_i^{\downarrow}\}$.

In RTLola, every stream and every expression has a concrete type, i.e., a single value type and a single pacing type. For streams, the concrete type is the type declared in the specification. An expression, however, does not have a type declaration. Thus, depending on the expression, the concrete type is not uniquely defined. The candidate set, i.e., the set of potential types is the abstract type of an expression.

Coercion

Remark 4.3.1 (Coercion in Specification Languages)**.** *A commonly used technique in type systems of programming languages is* coercion*. This techniques allows values of a certain type to be coerced into another type. For example, a value that might be a signed integer stored in 16 bits may be coerced into floating point number with 32 or 64 bits. There are different flavor of coercion, for number representation with (un-)signed integers and floating point numbers they mostly follow two rules:*

- *An unsigned integer can be coerced into a signed integer of equal bit-width, which can be coerced into a floating point number of equal bit-width.*
- *Any type of number can be coerced into a number of the same type with greater bit-width.*

The rules follow the intuition that a coercion from type T to T' is allowed if T' subsumes T in that T' can represent every value T can represent. So the rationale behind the first rule is thus $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{R}$*, and the rationale behind the second rule is that whatever fits in a container also fits in a larger container.*

These two intuitive rules need to be balanced with a set of unintuitive exceptions:

- *There are unsigned numbers that cannot be coerced into a signed integer. The integer constant* 10^{19} *fits into a* $UInt(64)$*, does not fit into an* $Int(64)$*, and does fit into a* $Float(64)$*.*
- *Not every integer can be accurately represented as a floating point number of the same width.*

In summary, neither coercion to floats, nor to integers is a guaranteed success. For a specification language, the goal is to have unambiguous semantics, so operations with the potential to fail at runtime pose a problem. Allowing for implicit coercion disregarding the issue can result in unexpected values during runtime. A remedy might be declaring the

coercion as fallible and requiring the specifier to declare a default value. As a result, the coercion is no longer an entirely implicit process, but a safe one. Since this is the major maxim behind the design of RTLola*, the language requires explicit casting for any type conversion. The only implicit coercion is the infallible bit expansion.*

Definition 15 (Abstract Types)

An *abstract type* summarizes the potential types an expression can assume. **Def.** Abstract Type

$$\widetilde{VT} := \{\emptyset, \{Bool\}, \{Float(64)\}, \{Float(32), Float(64)\}\}$$
$$\cup \bigcup_{i=4}^{7} \{\{UInt(y) | \{8,16,32,64\} \ni y \geq 2^i\}, \{Int(y) | \{8,16,32,64\} \ni y \geq 2^i\}\}$$

The abstract types regarding pacing types denote an upper bound on the evaluation frequency, less frequent evaluations are still an option. So if an expression has pacing type $\pi = 4$, it can be evaluated four times a second, but fewer evaluations are also possible. Similarly, the event type represents a minimal set of dependencies of an expression. If the expression is a part of a larger expression, the set may grow when climbing up the AST, but not diminish.

$$\widetilde{ET} := ET$$

$$\widetilde{PT} := PT$$

Throughout the rest of this thesis, we will indicate abstract types with a tilde over the name. Moreover, ι always represent event types and π represents periodic types. On top of that, σ denotes the pacing types and τ denotes value types.

4.3.1. Type Lattice

The type checking procedure of an RTLola specification is as agnostic of actual types as possible. It thus works on an abstract type lattice. It solely relies on the compatibility of abstract types, expressed by the means of abstract meet functions. After defining them, details about the type system can be ignored, allowing the type system to be changed or extended without modifying the type checking procedure.

Values types are explicit sets of potential candidate types. Thus, the most constrained set is the empty set denoting a contradiction. This is the least element in the lattice. As a result, the meet operation is the intersection of two candidate sets.

Definition 16 (Value Meet)

The value meet reduces two sets of candidate types to the intersection thereof.

$$\sqcap_{VT} := \cap$$

Lemma 4. *$(\widetilde{VT}, \sqcap_{VT})$ is a meet semi-lattice.*

Proof. One can easily see that the set intersection is defined on all pairs of elements in $\widetilde{VT}$ and itself member of $\widetilde{VT}$. Moreover, $\cap$ is associative, commutative and idempotent. □

Contrary to the value meet, the event meet is the union of two sets. This is because — intuitively —- the more elements are in an event type, the more rarely the expression gets evaluated.

Definition 17 (Event Meet)

The event meet reduces two sets of candidate sets to the union thereof, so the resulting type represents dependencies to all streams that were in either set.

$$\sqcap_{ET} := \cup$$

Lemma 5. *$(\widetilde{ET}, \sqcap_{ET})$ is a meet semi-lattice.*

Proof. By definition of $\widetilde{ET} = ET = 2^{Stream^{\downarrow}}$ the set union of two types is also part of $\widetilde{ET}$. Moreover, $\cup$ is associative, commutative, and idempotent. □

The periodic type declares the maximal frequency in which an expression can be evaluated. The meet thus needs to be a lower frequency that divides both operands.

Definition 18 (Periodic Meet)

The periodic meet is the greatest common divisor of both operands.

$$\widetilde{\pi_1} \sqcap_{PT} \widetilde{\pi_2} := \gcd(\widetilde{\pi_1}, \widetilde{\pi_1})$$

Lemma 6. *$(PT, \sqcap_{PT})$ is a meet semi-lattice.*

Proof. We first show that $\widetilde{\pi_1} \sqcap_{PT} \widetilde{\pi_2} = \gcd(\widetilde{\pi_1}, \widetilde{\pi_2}) \in \widetilde{PT}$.

By Definition 15:

$$\widetilde{PT} = \{p \mid p \in \mathbb{N} \wedge (\gcd(P^{spec}) \mid p) \wedge (p \mid \operatorname{lcm}(P^{spec}))\}$$

Thus,

$$\begin{aligned}
&\exists k_1, k_2 \in \mathbb{N}\colon \widetilde{\pi_1} = k_1 \cdot \gcd(P^{spec}) \wedge \widetilde{\pi_2} = k_2 \cdot \gcd(P^{spec}) \\
\implies &\gcd(k_1 \gcd(P^{spec}), k_2 \gcd(P^{spec})) = \gcd(P^{spec}) \gcd(k_1, k_2) \\
\implies &\gcd(P^{spec}) \mid \gcd(k_1 \gcd(P^{spec}), k_2 \gcd(P^{spec})) \\
\implies &\gcd(P^{spec}) \mid \gcd(\widetilde{\pi_1}, \widetilde{\pi_2})
\end{aligned}$$

Since $\widetilde{\pi_1} \in \widetilde{PT}$ we know $\exists k_1 \in \mathbb{N}\colon k_1 \widetilde{\pi_1} = \text{lcm}(P^{spec})$. By definition of gcd:

$$\exists k_1' \in \mathbb{N}\colon k_1' \,\text{gcd}(\widetilde{\pi_1}, \widetilde{\pi_2}) = \widetilde{\pi_2}$$

Thus:

$$\begin{aligned} & k_1' \,\text{gcd}(\widetilde{\pi_1}, \widetilde{\pi_2}) = \widetilde{\pi_2} \\ \implies & k_1 k_1' \,\text{gcd}(\widetilde{\pi_1}, \widetilde{\pi_2}) = k_1 \widetilde{\pi_2} \\ \iff & k_1 k_1' \,\text{gcd}(\widetilde{\pi_1}, \widetilde{\pi_2}) = \text{lcm}(P^{spec}) \\ \iff & \text{gcd}(\widetilde{\pi_1}, \widetilde{\pi_2}) \,\Big|\, \text{lcm}(P^{spec}) \end{aligned}$$

Since, $\text{gcd}(\widetilde{\pi_1}, \widetilde{\pi_2}) \in \mathbb{N}$, we can conclude $\text{gcd}(\widetilde{\pi_1}, \widetilde{\pi_2}) \in \widetilde{PT}$.

It remains to be shown that gcd is commutative and associative. Consider $\text{gcd}(a, b)$. By the fundamental theorem of arithmetic, we know that a and b can be represented as the product of primes numbers. Let $Primes(x)$ be the set of prime numbers in the prime number decomposition of x and $e_{p,x}$ the respective exponent such that for $x = a$ and $x = b$:

$$a = \prod_{p \in Primes(a)} p^{e_{p,a}} \qquad\qquad b = \prod_{p \in Primes(b)} p^{e_{p,b}}$$

We know:

$$\text{gcd}(a, b) = \prod_{\substack{p \in Primes(a) \\ \cup Primes(b)}} p^{\max(e_{p,a}, e_{p,b})}$$

Therefore, the associativity, commutativity, and idempotency follows from the associativity, commutativity, and idempotency of max, concluding the proof. □

Optional Values

The last addendum to the type system are *optional types*. They are a disjunctive monadic structure that either represents *something*, which wraps the value in a monad, or *nothing*. These kind of values are commonly found in functional programming languages like Haskell or Standard ML. Modern imperative languages like rust and swift increasingly opt for optional types as well to prevent the notorious *null pointer exceptions (NPE)*, a consequence of what Tony Hoare famously called his "Billion Dollar Mistake"[2]. NPE occur when a process calls a sub-routine that ought to produce a value. If the sub-routine fails, it silently returns an invalid value ("*null*"). The process now proceeds with its computation until it tries to access the value, which crashes the system. There can be a wide temporal disparity between the point of failure and the crash of the system, rendering the debug process excruciatingly cumbersome.

The whole conundrum can be mitigated by raising awareness that the sub-routine might fail. While infallible functions produce a value of type T, fallible ones produce

[2] https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/

a value of type $Opt\langle T\rangle$. This forces the programmer to consider the possibility that no value was produced. In a safety-critical language like RTLola, this paradigm constitutes a formidable choice for how to deal with fallible stream accesses. For this reason, there are two kind of expressions accessing a stream. The synchronous access cannot fail because the type system ensures that accesser and accessee have compatible timing. On the other hand, the sample and hold expression provides a way to access a stream with incompatible timing. Since there is no way to ensure that the access succeeds, the expression produces an optional value.

In its core, the type system of RTLola supplies the creation and destruction of optional values via stream accesses and default values, respectively. Thus, incorporating optional types in the type system requires the addition of an optional variant of each existing value type. The meet operation for optional types then refers back to the meet for value types.

Definition 19 (Type Lattice)

The full type system of RTLola is the meet-semilattice $(\widetilde{VT} \cup Opt\langle\widetilde{VT}\rangle \cup \widetilde{PT} \cup \widetilde{ET} \cup \{\bot\}, \sqcap)$ with the following meet operation:

$$\widetilde{\tau_1} \sqcap \widetilde{\tau_2} := \begin{cases} \widetilde{\tau_1} \sqcap_{VT} \widetilde{\tau_2} & \text{if } \widetilde{\tau_1}, \widetilde{\tau_2} \in VT \\ Opt\langle \widetilde{\tau_1}' \sqcap_{VT} \widetilde{\tau_2} \rangle' & \text{if } Opt\langle \widetilde{\tau_1}' \rangle = \widetilde{\tau_1} \wedge Opt\langle \widetilde{\tau_1}' \rangle = \widetilde{\tau_1} \\ \widetilde{\tau_1} \sqcap_{PT} \widetilde{\tau_2} & \text{if } \widetilde{\tau_1}, \widetilde{\tau_2} \in PT \\ \widetilde{\tau_1} \sqcap_{ET} \widetilde{\tau_2} & \text{if } \widetilde{\tau_1}, \widetilde{\tau_2} \in ET \\ \bot & \text{otherwise} \end{cases}$$

Proposition 7. *$(\widetilde{VT} \cup Opt\langle\widetilde{VT}\rangle \cup \widetilde{PT} \cup \widetilde{ET} \cup \{\bot\}, \sqcap)$ is a meet-semilattice.*

Proof. $\sqcap$ inherits associativity, commutativity, and idempotency from $\sqcap_{VT}, \sqcap_{ET}$, and $\sqcap_{PT}$, i.e., Lemma 4, 5, and 6, respectively. The closedness property of the lattice under $\sqcap$ follows from two points. Within a sub-lattice, the closedness follows from the closedness of the respective sub-lattice. The meet of two types from different sub-lattices follows from the inclusion of $\bot$ as fall-back. □

4.3.2. Type Checking

The goal of the type check is to find inconsistencies in the specification. For example, if an output stream is declared as an integer but the expression always yields a boolean value, something went wrong. Similarly, a trigger targeting a stream that yields numeric values constitutes a type error since trigger conditions are required to be boolean. For these checks, we define a type validity relation $\models$ where $\widetilde{\sigma}, \widetilde{\tau} \models e$ means that the abstract stream and value type $\widetilde{\sigma}$ and $\widetilde{\tau}$ model the expression e. A specification

is then only valid if there are valid pacing and value types for each stream and trigger. Otherwise, the specification is contradictory and cannot be monitored.

The search for valid types is based on inference rules. They are defined for streams and expressions and take type annotations into account. For this, the generalization function $lift\colon VT \to \widetilde{VT}$ is used. It transforms a concrete value type into an abstract value type while imposing the least amount of restriction.

$$lift(\tau) := \begin{cases} \{Bool\} & \text{if} \quad \tau = Bool \\ \{Xz \mid \{8,16,32,64\} \ni z \geq y\} & \text{if} \quad \tau = Xy, y \in \{8,16,32,64\}, \\ & \quad X \in \{Int, UInt, Float\} \end{cases}$$

One can easily see that the result of *lift* is indeed always a valid abstract type and always contains τ, i.e. $\tau \in lift(\tau)$.

The inference rule for input streams then asserts that the value type is compatible with the declared type and that the pacing type is exactly the singleton set of the stream itself.

$$\frac{\widetilde{\tau} \sqsubseteq lift(T_i^{\downarrow}) \quad \widetilde{\sigma} = \{s_i^{\downarrow}\}}{\widetilde{\sigma}, \widetilde{\tau} \models s_i^{\downarrow}} \tag{4.3}$$

Output streams reflect the types that are valid for their stream expression, with the additional constraint that it needs to respect annotated types. If an evaluation frequency is declared, the pacing type needs to be a periodic one; otherwise it needs to be an event type.

$$\frac{s_i^{\uparrow}.ext = \bot \quad \widetilde{\tau} \sqsubseteq lift(T_i^{\uparrow}) \sqcap \widetilde{\tau'} \quad \widetilde{\iota}, \widetilde{\tau'} \models s_i^{\uparrow}.expr}{\widetilde{\iota}, \widetilde{\tau} \models s_i^{\uparrow}}$$

$$\frac{\widetilde{\pi} \sqsubseteq s_i^{\uparrow}.ext \quad \widetilde{\tau} \sqsubseteq lift(T_i^{\uparrow}) \sqcap \widetilde{\tau'} \quad \widetilde{\sigma}, \widetilde{\tau'} \models s_i^{\uparrow}.expr}{\widetilde{\pi}, \widetilde{\tau} \models s_i^{\uparrow}}$$

For triggers, the evaluation target needs to yield a boolean value.

$$\frac{\{Bool\} \sqsubseteq \widetilde{\tau} \quad \widetilde{\sigma}, \widetilde{\tau} \models s_i^{!}.tar}{\widetilde{\sigma}, \widetilde{\tau} \models s_i^{!}}$$

Next are the inference rules for expression, starting with synchronous lookups. These expressions bind the timing of a stream to the one of the accessed stream. This translates to a respective constraint in the inference rule: the accessor's type needs to be a more concrete type than the accessee's type.

$$\frac{\widetilde{\sigma'}, \widetilde{\tau'} \models s_i^{-} \quad \widetilde{\sigma} \sqsubseteq \widetilde{\sigma'} \quad \widetilde{\tau} \sqsubseteq \widetilde{\tau'}}{\widetilde{\sigma}, \widetilde{\tau} \models Sync(s_i^{-})} \tag{4.4}$$

An offset expression behaves similar. The two differences are that the access might fail because the accessed stream did not produce enough values yet. Thus, the resulting value type is an optional type. Moreover, the offset needs to be a natural number.

$$\frac{\widetilde{\sigma}', \widetilde{\tau}' \models s_i^- \quad \widetilde{\sigma} \sqsubseteq \widetilde{\sigma}' \quad \widetilde{\tau} \sqsubseteq \mathit{Opt}\langle\widetilde{\tau}'\rangle \quad n \in \mathbb{N}}{\widetilde{\sigma}, \widetilde{\tau} \models \mathit{Offset}(s_i^-, n)}$$

The sample and hold access can also fail and thus results in an optional value. However, opposed to the other kinds of accesses, it decouples the timing of accessor and accessee.

$$\frac{\widetilde{\sigma}', \widetilde{\tau}' \models s_i^- \quad \widetilde{\tau} \sqsubseteq \mathit{Opt}\langle\widetilde{\tau}'\rangle}{\widetilde{\sigma}, \widetilde{\tau} \models \mathit{Hold}(s_i^-)}$$

Default expressions relieve the specification of any optional types by providing a default value if the lookup has failed. In addition to that, the types of both subexpressions need to be compatible. The meet of both types constitutes the type of the default expression.

$$\frac{\widetilde{\sigma_1}, \widetilde{\tau_1} \models e_1 \quad \widetilde{\sigma_2}, \widetilde{\tau_2} \models e_2 \quad \widetilde{\sigma_1} = \mathit{Opt}\langle\widetilde{\sigma_1}'\rangle \quad \widetilde{\sigma} \sqsubseteq \widetilde{\sigma_1}' \sqcap \widetilde{\sigma_2} \quad \widetilde{\tau} \sqsubseteq \widetilde{\tau_1} \sqcap \widetilde{\tau_2}}{\widetilde{\sigma}, \widetilde{\tau} \models \mathit{Default}(e_1, e_2)} \tag{4.5}$$

Unsurprisingly, Function expressions heavily depend on the function itself. The function's type acts as explicit type annotations for the passed arguments. The generalized return type of the function then needs to match the expression's type.

$$\frac{f\colon T_1 \times \cdots \times T_n \to T \quad \widetilde{\tau} \sqsubseteq \mathit{lift}(T) \quad \forall i\colon \widetilde{\sigma_i}, \widetilde{\tau_i} \models a_i \quad \forall i\colon \widetilde{\tau_1} \sqsubseteq \mathit{lift}(T_i) \quad \widetilde{\sigma} \sqsubseteq \widetilde{\sigma_1} \sqcap \ldots \sqcap \widetilde{\sigma_n}}{\widetilde{\sigma}, \widetilde{\tau} \models \mathit{Func}(f, a_1, \ldots, a_n)} \tag{4.6}$$

Lastly, aggregation expressions impose restrictions on the value type of the accessed stream and the result of the expression according to the typing of the aggregation function. Moreover, the aggregation expression itself needs to be of periodic type.

The rationale behind this is as follows: The evaluation of a sliding window expression is inherently more involved than other computations barring excessive nesting of expensive operations. Thus, disallowing these computations in the uncontrollable realm of event-based streams can impact the performance of the evaluator significantly. A side-effect of this decision is that it also enables the determination of a memory bound for an evaluator. Without the restriction, the required memory is unbounded. The details of the efficient computation follow in Section 5.2.

→ Sec. 5.2, Page 57

$$\frac{\delta \in \mathbb{N} \quad \gamma\colon T_a^* \to T_r \quad \tau \sqsubseteq \mathit{lift}(T_r) \quad \widetilde{\sigma}', \widetilde{\tau}' \models s_i^- \quad \widetilde{\tau}' \sqsubseteq \mathit{lift}(T_a)}{\widetilde{\pi}, \widetilde{\tau} \models \mathit{Window}(s_i^-, \delta, \gamma)} \tag{4.7}$$

```
input a: Int32
input b: Float64
output x: Float64 := multiply(a, b.hold.defaults(to: 99))
trigger x "Type checks prevent errors before deployment!"
```

Figure 4.6.: An RTLola specification that has valid types up to the trigger, which requires a boolean value and provides a floating point number.

Definition 20 (Type Validity)

A specification has *valid types* if and only if for every stream and trigger there is a non-contradictory value and pacing type. **Def.** Type Validity

$$\forall s_i^- \exists \widetilde{\sigma}, \widetilde{\tau} \colon \widetilde{\sigma}, \widetilde{\tau} \models s_i^- \wedge \widetilde{\sigma} \neq \bot \wedge \widetilde{\tau} \neq \bot$$

Example 4.3.1 (Type Checking)**.** Consider the specification in Figure 4.6, which is a simplification of the specification from Example 4.1.1. → Sec. 4.1, Page 19

The only viable choices for type for the input streams are as follows:

$$\{s_1^{\downarrow}\}, \{Int(32), Int(64)\} \models s_1^{\downarrow}$$

$$\{s_2^{\downarrow}\}, \{Float(64)\} \models s_2^{\downarrow}$$

The type check of the output stream starts with the stream accesses. While the access to `a` is synchronous and thus imposes a restriction on the pacing type, the access to `b` is a sample and hold expression. As such, there is no immediate requirement on the pacing type. However, the application of the `multiply` function requires that both arguments have compatible stream types. Therefore, we already choose a suitable stream type to prevent backtracking.

$$\{s_1^{\uparrow}\}, \{Int(32), Int(64)\} \models Sync(s_1^{\downarrow})$$

$$\{s_1^{\uparrow}\}, \{Float(64)\} \models Hold(s_2^{\downarrow})$$

$$\{s_1^{\uparrow}\}, \{Float(64)\} \models Func(f, Sync(s_1^{\downarrow}), Hold(s_2^{\downarrow}))$$

$$f : Int(32) \times Float(64) \rightarrow Float(64)$$

$$\{s_1^{\uparrow}\}, \{Float(64)\} \models s_1^{\uparrow}$$

The types for $s_1^{\uparrow}$ are not a choice but enforced by the respective inference rule. As a result, the only viable choice for the value type of the trigger is $\{Float(32)\} \sqcap \{Bool\} = \bot$, rendering the specification invalid; this is expected because a trigger needs a boolean condition which `x` does not provide.

The specification can be fixed by introducing another output stream `y` of type *Bool* that accesses `x` synchronously and applies a boolean condition on it. The target of the trigger then needs to be changed to `y`. △

Remark 4.3.2 (Type Choices)**.** *The example already indicates that the types for each stream are not always uniquely defined. The resolution of this ambiguity is to require that the final type assignment is always the least restrictive viable choice. This is a unique solution because choices are always comparable, which means, all choices are in the same sub-lattice:*

- *A stream cannot declare an optional type, so the $\sqsubseteq$ relation on value types can always be applied.*
- *An output stream is periodic iff it declares an evaluation frequency. Therefore, a periodic and an event-based type can never be viable at the same time. Thus, all viable types are either periodic or event-based.*
- *The contradictory type ($\bot$) cannot be part of a valid specification.*

The type analysis concludes the series of static criteria for RTLola specifications. The following definition summarizes the criteria for valid specifications.

Definition 21 (Specification Validity)

Def. Valid Specification

A RTLola specification is *valid* iff it satisfies the following three criteria on its syntax, dependency graph, and types:

→ Sec. 4.1, Page 19

- Syntactic validity according to Definition 7

→ Sec. 4.2, Page 23

- Well-formedness according to Definition 10

→ Sec. 4.3, Page 35

- Type validity according to Definition 20

For the type system we introduced optional types. Fallible stream accesses like offset expressions and sample and hold expressions produce these type. However, the following lemma shows that a valid stream expression cannot have an optional type. This means that a stream expression cannot yield an optional value while sub-expressions can. The only possibility to remove optional types is the usage of default expressions. Thus, intuitively, every fallible stream expression is encompassed in a default expression.

Lemma 8 (Absence of Optional Output Types)**.** *In a valid* RTLola *specification, a stream expression cannot have an optional value type.*

$$\forall s_i^{\uparrow} \in \mathit{Stream}^{\uparrow} : \widetilde{\sigma}, \widetilde{\tau} \models s_i^{\uparrow}.\mathit{expr} \implies \forall \widetilde{\tau}' : \widetilde{\tau} \neq \mathit{Opt}\langle \widetilde{\tau}' \rangle.$$

Proof. By definition of type validity, there are valid types $\widetilde{\sigma}, \widetilde{\tau}$ with $\widetilde{\sigma}, \widetilde{\tau} \models s_i^{\uparrow}.\mathit{expr}$. Here, $\widetilde{\tau} = \mathit{lift}(\tau_d) \sqcap \widetilde{\tau_e}$ is the meet of the lift of the concrete declared type τ_d and the type of the expression, i.e., $\widetilde{\sigma}, \widetilde{\tau_e} \models s_i^{\uparrow}.\mathit{expr}$. Further, we know $\widetilde{\tau} \neq \bot$ because the specification has valid types. By definition of $\sqcap$, $\widetilde{\tau}$ can only be an optional type if both $\mathit{lift}(\tau_d)$ and $\widetilde{\tau_e}$ are optional types. Declaring optional types is not syntactically valid, rendering τ_d non-optional. As a result, $\mathit{lift}(\tau_d)$ cannot be optional as well. □

Corollary 9. *In a valid* RTLola *specification, any fallible expression, i.e., offset or sample and hold expression, is encompassed in a default expression, where it is a sub-expression of the first argument.*

Proof. The encompassment follows from Lemma 8 and the inference rules for RTLola expressions: Default expressions are the only expressions that can destruct optional types. The fact that it has to be the first argument follows from the inference rule for default values (Equation 4.5) that destructs only the optional type of the first argument. → Sec. 4.3, Page 34
□

4.4. Semantics

The preceding sections, the examples argued the semantics of RTLola on an intuitive level. While this suffices to get the general gist of the language, the lack of formal meaning can hinder a deployment on safety critical devices. So this section introduces the formal semantics of RTLola. Hereby, we first introduce RTLola's two-fold concept of time: it is related to incoming events and the real wall-clock time. Both aspects can trigger evaluations of streams independent of each other, so the evaluation of periodic streams does not rely on the arrival of certain input values and events can arrive at arbitrary points in time. The resulting evaluation process is split into multiple phases based on the evaluation layers computed using the dependency graph.

Note that the formal semantics disregards triggers. They do not have an expression and merely access a boolean stream synchronously, there is no value to be computed and the memory configuration does not change.

The goal of the semantics is to construct a model for the specification, i.e., an infinite sequence of values for each stream, disregarding the evident impracticality. Any realization of the semantics can neither store nor even construct any infinite sequence. For this reason, we will identify a criterion on a practical realization that renders them "close enough to the actual semantics", so they reflect them properly.

4.4.1. Handling Time

The split of pacing types into periodic and event-based types indicates that there are two criteria that start the evaluation of streams. Periodic streams follow a static *Schedule* that contains points in time when the stream needs to be evaluated. A stream with evaluation frequency 5Hz, for example, has a schedule that starts an evaluation every 200ms. Successively adding other streams and merging the schedules yields a specification-global schedule that repeats after its *hyper-period*. The hyper-period is the inverse of the gcd of all evaluation frequencies, or — equivalently — the lcm of all evaluation periods $\text{lcm}(\{p^{-1} \mid p \in P^{spec}\})$.

Schedule

Hyper-Period

On the contrary, the points in time when event-based streams need to be evaluated is statically undetermined; the monitor has no information on when input values arrive and trigger an evaluation. The semantics, however, are a theoretical construct and thus have access to all the information. Formally, events are of the following shape:

Definition 22 (Event Sequence)

Def. Event Sequence
Def. Event

The input *event sequence* is an infinite sequence of events $e_1 e_2 \cdots \in \mathcal{E}^\omega$. Each *event* $e_i \in \mathcal{E} = (T_j^\downarrow \cup \{\bot\})_{j \le n^\downarrow} \times \mathbb{R}_+$ itself is a sequence of values $(v_j)_{j \le n^\downarrow}$, each value belonging to an input stream. An undefined ($\bot$) value v_j indicates that the stream $s_j^\downarrow$ did not receive a new value. Otherwise, v_j is in the domain of $s_j^\downarrow$. Moreover, the event is coupled with a real-valued timestamp. We abbreviate $e_i = (v_i, t_i)$ and $e_i.time = t_i, e_i.data = v_i$.

The annotated timestamp enables a merge of event arrival times and the schedule. This results in a countably infinite set of timestamps on which an evaluation needs to take place.

Definition 23 (Relevant Timestamps)

Def. Relevant Timestamp

The sequence of *relevant timestamps* $(\hat{t}_i)_{i \in \mathbb{N}}$ is comprised of all timestamps when event-based or periodic streams need to be evaluated. For each relevant timestamp $\hat{t}_i$, the following holds:

$$\exists j \in \mathbb{N} \colon e_j.time = \hat{t}_i \vee \exists s_j^\uparrow \in Stream^\uparrow, k \in \mathbb{N} \colon s_j^\uparrow.ext \neq \bot \wedge s_j^\uparrow.ext^{-1} \cdot k = \hat{t}_i$$

Moreover, the sequence needs to be monotonically increasing, i.e., $\forall i \colon \hat{t}_i < \hat{t}_{i+1}$.

Note that the arrival of an event can coincide with a deadline in the overall schedule. This does only produce a single relevant timestamp due to the strict monotonicity requirement.

Example 4.4.1. Consider the following simple specification.

```
input a: Int8
input b: Int8
output v: Int8 := a
output w: Int8 := a + b
output x: Int8 @10Hz := 800
output y: Int8 @5Hz := 85
```

The output streams `v` and `w` have event type $\{a\}$ and $\{b\}$, respectively. The output streams `x` and `y` have periodic types 10 and 5, which corresponds to an evaluation period of $(10\text{Hz})^{-1} = .1\text{s}$ and $(5\text{Hz})^{-1} = .2\text{s}$, respectively. The schedule of the entire specification thus has a hyper period of 200ms with two entries: $\{x\}$ after 100ms and $\{x, y\}$ after 200ms. Now assume the event sequence contains four events:

$$e_1 = ((1, \bot), 0.02)$$
$$e_2 = ((2, 3), 0.11)$$
$$e_3 = ((\bot, 4), 0.26)$$
$$e_4 = ((5, 6), 0.4)$$

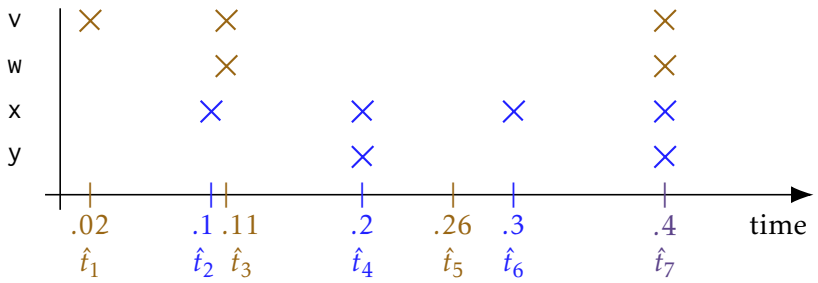


Figure 4.7.: Illustration of the real time axis for a given event sequence and specification. The evaluation of event streams (gold) `v` and `w` follows no pattern; `x` and `y` are evaluated periodically (blue) after .1s and .2s, respectively. $\hat{t}_7$ is both periodic and event-based.

As a result, the first seven entries of the sequence of relevant timestamps are 0.02, 0.1, 0.11, 0.2, 0.26, 0.3, 0.4.

Figure 4.7 illustrates the real time axis and which streams are affected by each relevant timestamp. Streams `x` and `y` are evaluated at regular intervals whereas `v` and `w` depend on the irregular events. At relevant timestamp $\hat{t}_5 = 0.26$, only `b` receives a new value which does not trigger any evaluation: `v` depends solely on `a` and `w` requires new values for both `a` and `b`.

△

4.4.2. The Evaluation Process

Section 4.2.1 outlined the notion of an evaluation model for RTLola specifications. The evaluation model is an infinite sequence of values for each stream. It contains the values every stream assumes "after" the infinite sequence of events and time has passed. Formally, we represent the model as a memory configuration, and access it with a function mapping streams and discrete time indicators to values. The time indicators refer to relevant timestamps.

→ Sec. 4.2, Page 22

Definition 24 (Memory Configuration)

A *memory configuration* $M \in \mathcal{MC}$ contains the history of all values that have been observed as inputs or computed as outputs. A memory access function μ constitutes the interface between the abstract memory configuration and concrete values.

Def. Memory Config

$$\mu \colon \mathcal{MC} \to (\mathit{Stream}^{\downarrow} \times \mathit{Stream}^{\uparrow}) \to \mathbb{N} \to (\bigcup_{i=1}^{n^{\downarrow}} T_i^{\downarrow} \cup \bigcup_{i=1}^{n^{\uparrow}} T_i^{\uparrow} \cup \{\#\})$$

Hereby, # denotes an invalid value.

Intuitively, $\mu(M)(s_i^{\downarrow})(5)$ refers to the fifth value input stream $s_i^{\downarrow}$ ever received.

Remark 4.4.1 (Structure of Memory Configurations)**.** *The definition of a memory configuration leaves the structure of a configuration entirely open. This allows for an easy exchange of representations of memory configurations as only the output of the access function μ is relevant. More concretely, the semantics defined here require an unbounded amount of memory. Any realization thereof does not have this luxury and needs to cope with memory limitations.*

→ Sec. 4.2, Page 26

The semantics are now defined inductively, primarily on the sequence of relevant timestamps, and secondarily on the evaluation order (Definition 12). That is, the initial memory configuration is M^0; M^{i+1} for $i > 0$ is the memory configuration after evaluating relevant timestamp $\hat{t}_i$. In-between two relevant timestamps, we define several sub-configurations: M^i_λ denotes the memory configuration after evaluating $\hat{t}_{i-1}$ fully, and the first λ evaluation layers for $\hat{t}_i$.

Evaluation Step

For each *evaluation step*, i.e. relevant timestamp and evaluation layer, the *active* predicate determines whether a stream needs to be evaluated. The first criterion for this is whether the current timestamp affects a stream. For this, $\exists s^{\uparrow}_j, k\colon s^{\uparrow}_j.ext \cdot k = t$ identifies a periodic deadline and $\exists j\colon e_j.time = \hat{t}_i$ the arrival of input data. Note that both criteria can hold for the same point in time.

Definition 25 (Stream Activation)

Assume $\hat{t}_i$ is induced by an event e_q. An event-based stream $s^{\uparrow}_j$ with event-type ι is *active* in evaluation layer λ if its e_q covers ι and $s^{\uparrow}_j$ is in layer λ.

Def. Stream Activation

$$active(s^{-}_j, i, \lambda) \equiv Layer(s^{-}_j) = \lambda \wedge \forall s^{\downarrow}_k \in \iota\colon e_k.data \neq \bot$$

If $\hat{t}_i$ is induced by a periodic deadline, a periodic stream $s^{\uparrow}_j$ with periodic type π is active in layer λ if its evaluation layer is λ and the period divides the current timestamp.

$$active(s^{\uparrow}_j, i, \lambda) \equiv Layer(s^{-}_j) = \lambda \wedge \pi \mid \hat{t}_i$$

Example 4.4.2. Recall the specification from Example 4.4.1. Input streams `a` and `b` are in layer 0, the output streams `v`, `w`, `x`, and `y` are in layer 1. The following table summarizes in which evaluation steps the streams are active.

$\hat{t}_i$	1		2		3		4		5		6		7	
λ	0	1	0	1	0	1	0	1	0	1	0	1	0	1
a	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗
b	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✓	✗
v	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗
w	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓
x	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✓	✗	✓
y	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓

△

The initial memory configuration M^0 does not contain any information and is thus comprised of # indicating a non-extant value. In subsequent steps, the *fempty* predicate indicates the first empty spot in the memory configuration, i.e., when $s_i^{\uparrow}$ has been active 8 times before, *fempty* yields 9.

$$fempty(M, s_j^-) := \min\{x \mid \mu(M)(s_j^-)(x) = \#\} \tag{4.8}$$

Definition 26 (RTLola Memory Semantics)

The memory semantics of RTLola are defined based on the desired output of μ. Initially, all values are invalid:

$$\forall n \in \mathbb{N}: \quad \mu(M_0)(s_j^-)(n) := \# \tag{4.9}$$

After that, assume relevant timestamp $\hat{t}_i$ with $i > 0$ and layer $\lambda = 0$:

$$\mu(M_i^0)(s_j^-)(x) := \begin{cases} \mu(M_{i-1})(s_j^-)(x) & \text{if } \neg active(s_j^-, i, 0) \vee x \neq fempty(M, s_j^-) \\ e_q.data[j'] & \text{if } s_j^- = s_{j'}^{\downarrow} \end{cases}$$

For $\lambda > 0$:

$$\mu(M_i^\lambda)(s_j^{\uparrow})(x) := \begin{cases} \mu(M_i^{\lambda-1})(s_j^-)(x) & \text{if } \neg active(s_j^-, i, \lambda) \vee x \neq fempty(M, s_j^-) \\ eval(M_i^{\lambda-1})(s_j^{\uparrow}.expr) & \text{otherwise} \end{cases}$$

Lastly, the new memory configuration is the configuration after evaluating the greatest layer.

$$M_{i+1} := M_i^{\lambda^*}$$

Intuitively, the evaluation starts with layer 0 containing only input streams. All values of inactive streams remain unchanged, as well as all entries of active input streams that are not the first empty value. The first empty value now contains the respective data from the event e_q. The evaluation then proceeds layer by layer, following the same general idea. The only difference is that rather than copying new values from the input events, the stream expression gets evaluated, which is done in the *eval* function defined later. This function evaluates an expression and returns the resulting value.

4.4.3. Expression Evaluation

The most non-standard aspect of expression evaluation in RTLola is stream access. These can declare a relative offset based on the time line of the accesses stream. For example, if a stream with evaluation frequency 10Hz accesses a stream with frequency

1Hz using an offset of −3, the accessed value is 3s old. The *r2a* function translates a relative offset o for an access to stream s^-_j at relevant timestamp $\hat{t}_i$ in layer λ into the absolute offset for an access within memory configuration M as follows:

$$r2a(M, s^-_j, i, \lambda, o) := fempty(M, s^-_j) - 1 - o + \mathbb{1}_{\exists \lambda' \in [\lambda, \lambda^*]\colon active(s^-_j, i, \lambda)} \tag{4.10}$$

Intuitively, the function first computes the first free value for s^-_j. Subtracting 1 yields the first non-# value, and subtracting the offset o yields the desired value granted no pseudo-evaluation (cf. Example 4.2.3) took place. If s^-_j needs to be evaluated in a greater layer, the semantics assume the evaluation already took place and compensates by reducing the offset by 1.

→ Sec. 4.2, Page 23

The evaluation of a sliding window expression $w_x = Window(s^-_j, \delta, \gamma)$ takes the arrival or computation time of values into account. It aggregates all values of s^-_j that occurred in the last δ seconds using γ. The number of relevant values for the window are thus:

$$inwin_{w_x}(i) := |\{\eta \in [0, \dots, i] \mid \hat{t}_\eta > \hat{t}_i - \delta \wedge \exists \lambda\colon active(s^-_j, \eta, \lambda)\}| \tag{4.11}$$

Note that the values for η represent the indices of relevant timestamps that qualify for the window and in which the window target was active. As such, they are neither relative, nor absolute offsets of qualifying values in M.

Definition 27 (RTLola Expression Semantics)

The evaluation of a stream expression of stream $s^\uparrow_k$ at relevant timestamp $\hat{t}_i$ is defined as:

$$eval_{s^\uparrow_k}(M)(Func(f, a_1, \dots, a_n)) := f(eval_{s^\uparrow_k}(M)(a_1), \dots, eval_{s^\uparrow_k}(M)(a_n))$$

$$eval_{s^\uparrow_k}(M)(Default(e, e')) := \begin{cases} eval_{s^\uparrow_k}(M)(e) & \text{if } eval_{s^\uparrow_k}(M)(e) \neq \# \wedge eval_{s^\uparrow_k}(M)(e) \neq \bot \\ eval_{s^\uparrow_k}(M)(e') & \text{otherwise} \end{cases}$$

$$eval_{s^\uparrow_k}(M)(Sync(s^-_j, n, d)) := \mu(M)(s^-_j)(r2a(M, s^-_j, i, Layer(s^\uparrow_k), 0))$$

$$eval_{s^\uparrow_k}(M)(Offset(s^-_j, n)) := \begin{cases} \mu(M)(s^-_j)(x) & \text{if } x = r2a(M, s^-_j, i, Layer(s^\uparrow_k), n) \wedge x \geq 0 \\ \# & \text{otherwise} \end{cases}$$

$$eval_{s^\uparrow_k}(M)(Hold(s^-_j)) := eval_{s^\uparrow_k}(M)(Offset(s^-_j, 0))$$

$$eval_{s^\uparrow_k}(M)(Window(s^-_j, \delta, \gamma)) := \gamma((\mu(M)(s^-_j)(r2a(M, s^-_j, i, \lambda, \eta)))_{0 \leq \eta \leq inwin_{Window(s^-_j, \delta, \gamma)}(i)})$$

The evaluation of function expressions requires to evaluate each argument and apply the function afterwards — nothing unexpected. Similarly straight-forward are default expressions. If the first argument yields a non-empty value, it becomes the result of the

expression. Otherwise the second argument is evaluated and returned; the type system ensures that this value cannot be empty. Synchronous stream accesses also utilize the type system: the pacing type of $s_k^{\uparrow}$ and s_j^{-} need to be compatible so s_j^{-} will receive a new value in the evaluation cycle. The access forces s_j^{-} to be in a lower evaluation layer than $s_k^{\uparrow}$, so the value was already computed and the access always succeeds. This assumption does not hold for offset expressions. If s_j^{-} did not produce at least n values, the access fails. As a result, the evaluation yields an empty value. While the type system restricts offset expressions by requiring compatible pacing types, the sample and hold expression merely requires the accessee to have at least one value. Semantically, it thus behaves equivalently to an offset expression with $n = 0$. Lastly, window expressions are the only expressions that are not time-agnostic. Note that γ takes an unbounded, statically undetermined number of values as argument because the evaluation frequency of s_j^{-} is potentially unknown. The evaluation computes how many values are relevant for the window, accesses all of them and aggregates them oldest to newest using γ.

The crucial property for expression evaluations is that every stream expression will always yield a non-empty value.

Theorem 10 (Valid Accesses)**.** *For any $i \in \mathbb{N}$ and any layer λ, the evaluation of a stream expression will always result in a valid value, i.e., the value is neither empty nor undefined. The latter would be the result of an invalid argument to the μ function.*

$$eval_{s_k^{\uparrow}}(M_i^{\lambda})(s_k^{\uparrow}.expr) \notin \{\#, \bot\}$$

Proof. By structural induction on the expression AST. Synchronous lookup, offset, sample and hold, and sliding window expressions constitute the base cases. We assume the specification is valid and can thus make use of the type system.

1) Consider a synchronous lookup, so $s_k^{\uparrow}.expr = Sync(s_j^{-})$.

The type system (Equation 4.4) provides knowledge that s_j^{-} has a compatible pacing. →Sec. 4.3, Page 33
Moreover, the definition of *Layer* (Definition 12) guarantees that $Layer(s_j^{-}) < Layer(s_k^{\uparrow})$. →Sec. 4.2, Page 26
Therefore, there is a layer $\lambda' < \lambda$ with $active(s_j^{-}, i, \lambda')$. While inactive streams carry their last value over, active streams get a new one. In layer 0, i.e., for input streams, this is a copy of a value of the current input event. By the type system (Equation 4.3) and the definition of *active*, the value is defined and thus non-empty. In a layer greater than 0, i.e., the stream in an output stream, it only has a valid value if this theorem is true. However, we can apply the argument recursively. This is well-founded by the strict decline of the evaluation layer. The last thing to consider is the offset passed to the memory access function, computed by *r2a*:

$$
\begin{aligned}
r2a(M_i^\lambda, s_j^-, i, \lambda, 0) &= fempty(M_i^\lambda, s_j^-) - 1 - 0 + \mathbb{1}_{\exists \lambda' \in [\lambda, \lambda^*]:\ active(s_j^-, i, \lambda)} \\
&= fempty(M_i^\lambda, s_j^-) - 1 - 0 + 0 && (Layer(s_j^-) < Layer(s_k^\uparrow)) \\
&= fempty(M_i^\lambda, s_j^-) - 1 \\
&\geq 1 - 1 = 0
\end{aligned}
$$

This derivation shows two things: the result of *r2a* is non-negative and strictly less than $fempty(M_i^\lambda, s_j^-)$. This suffices to conclude that the synchronous lookup always results in a valid value, it breaks when considering the next case: offset accesses.

3) Consider an offset expression, so $s_k^\uparrow.expr = Offset(s_j^-, n)$.

In former sections, this kind of stream access was referred to as "fallible". That is because $r2a(M_i^\lambda, s_j^-, i, \lambda, n)$ can be negative if n exceeds the number of values s_j^- already produced:

$$
\begin{aligned}
& r2a(M_i^\lambda, s_j^-, i, \lambda, n) && < 0 \\
\iff & fempty(M_i^\lambda, s_j^-) - 1 - n + \mathbb{1}_{\exists \lambda' \in [\lambda, \lambda^*]:\ active(s_j^-, i, \lambda)} && < 0 \\
\iff & fempty(M_i^\lambda, s_j^-) - 1 + \mathbb{1}_{\exists \lambda' \in [\lambda, \lambda^*]:\ active(s_j^-, i, \lambda)} && < n
\end{aligned}
$$

→ Sec. 4.3, Page 37

However, by Corollary 9, fallible accesses are surrounded by default expressions, preventing optional types. Thus, we can exempt offset lookups from the proof, granted default expressions always yield a valid value. The same reasoning applies to the sample and hold case, i.e., for $s_k^\uparrow.expr = Sample(s_j^-)$.

4) Consider a sliding window expression, so $s_k^\uparrow.expr = Window(s_j^-, \delta, \gamma)$.

The aggregation function γ aggregates an arbitrary number of values to an intermediate value. This especially includes the empty sequence for which the aggregation yields the neutral value ε_γ. All other stream accesses are necessarily valid accesses be-

→ Sec. 4.4, Page 42

cause the last $n = inwin_w(i)$ values are accesses. By its definition (Equation 4.11), the value is upper bounded by the amount of times *active* held. Yet again, the argument can be applied recursively, which is well-defined because a sliding window imposes a dependency in the dependency graph.

This concludes the base cases. The inductive cases consist of function and default expressions. The former is trivial: By induction all arguments yield a valid value. Thus,

→ Sec. 4.3, Page 34

by the type check for function expressions (Equation 4.6), the function is applicable for the arguments, resulting in a valid value. A default expression $e = Default(e_1, e_2)$

→ Sec. 4.3, Page 34

has two sub-expressions as argument. The type systems (Equation 4.6) allows the first argument to result in an optional value. However, the expression evaluation function replaces an invalid value with the result of evaluating the second argument. By Corol-

→ Sec. 4.3, Page 37

lary 9, either e is again encompassed in a default expression where it is the first argument, or it does yield a non-optional value. In the former case, the optional value is

passed upwards until reaching the respective default expression. In the latter case, by recursive application of this proof argument, the resulting value is valid. In both cases, recursion is well-defined because the abstract expression tree is finite and acyclic. Thus, the result of evaluating a default expression is a valid value. □

Corollary 11. *The semantics never requires an assignment of an empty or undefined value after the initialization.*

Proof. According to Definition 26, there are three kind of assignment: →Sec. 4.4, Page 41

- A replication of an old value if the stream is not active in the current evaluation phase. An uninitialized stream then remains uninitialized. For initialized streams, the result of one of the other kinds of assignments gets carried over.
- An input event value is assigned if the stream is active and the layer is 0. The assignment further requires that the target stream $s_i^{\downarrow}$ is an input stream. Thus, its pacing type is the event-type $\iota = \{s_i^{\downarrow}\}$ by type validity (Definition 20) and the inference rule for input streams (Equation 4.3). Lastly, the definition of *active* requires that the input event is defined for all streams in the event type. →Sec. 4.3, Page 35 →Sec. 4.3, Page 33
- The result of an expression evaluation is assigned. By Theorem 10, this is a valid value.

□

4.4.4. Evaluation Model

We will now re-instate the result of D'Angelo [47] concerning the unique existence of an evaluation model for Lola. This formalizes the intuition given in Section 4.2.1. →Sec. 4.2, Page 22

The semantics of an RTLola specification are defined solely by the memory access function μ. This is an implicit notion of the evaluation model. An explicit notion of the model is a collection of sequences representing all values any stream every got. In Lola, every stream is evaluated every time a new input arrives, resulting in infinite models for all streams. In RTLola, however, each sequence is separately potentially infinite: periodic streams are active infinitely often, so their model is always infinite. Event-based streams with event-type ι are only infinite if there is an infinite sub-sequence of events, where each event covers ι entirely.

The evaluation model for a stream in an RTLola specification is thus the sequence of values that μ is required to yield when the stream is active. The model is uniquely defined if μ is uniquely defined for each point in time.

Proposition 12. *The memory access function μ is uniquely defined.*

Proof Sketch. Theorem 10 shows that any stream evaluation always yields at least one valid value. The uniqueness follows trivially from Definition 26 regarding the memory semantics and the evaluation of expressions (Definition 27): The values for input →Sec. 4.4, Page 43 →Sec. 4.4, Page 41 →Sec. 4.4, Page 42

streams come from input events, which are unique by the definition of events (Definition 22).

→ Sec. 4.4, Page 38

Output stream values come from *eval*. Their uniqueness follows by induction on the sequence of memory configurations complying with the requirements imposed on μ.

Function Expressions yield unique values because the respective function yields a unique value.

Default, Offset, Sample and Hold, and Synchronous Expressions refer to value "older" values of the memory semantics and are thus unique by induction.

Sliding Window expressions aggregate "older" values of the memory semantics and the aggregation function is uniquely defined, therefore the window expression is as well.

□

4.4.5. Monitoring

With the fully defined semantics of RTLola specifications, we can finally define a *monitor*. Intuitively, a monitor for a specification takes event values and reports whether the specification is violated or not. As opposed to the semantics, this is an entirely finite concept. While the monitor can potentially run for an infinite amount of time, it only has access to a finite prefix of the events and ran for a finite number of hyper-periods of the specification.

Moreover, the semantics referred to the memory access function μ, not requiring insights into the concrete memory configuration. The requirements on the monitor are similar. It does, however, require a timestamp indicating the termination time. This time can differ from the timestamp of the last event in the input sequence. While the termination time has no influence on the general monitoring process, it is required for checking compliance of the monitor to the semantics of the specification.

Definition 28 (Monitor)

A monitor $\mathcal{M}_\Phi$ for a specification Φ transforms a finite prefix of events and the termination time into a mapping from streams to values.

$$\mathcal{M}_\Phi : \mathcal{E}^* \times \mathbb{R}_+ \to \mathit{Stream}^\uparrow \to \bigcup_{i=1}^{n^\uparrow} T_i^\uparrow$$

→ Sec. 4.4, Page 39

Recall the type of the memory access function μ from Definition 24:

$$\mu : \mathcal{MC} \to (\mathit{Stream}^\downarrow \times \mathit{Stream}^\uparrow) \to \mathbb{N} \to (\bigcup_{i=1}^{n^\downarrow} T_i^\downarrow \cup \bigcup_{i=1}^{n^\uparrow} T_i^\uparrow \cup \{\#\})$$

The type of of the monitor output $\mathcal{M}_\Phi(e_1 \dots e_k, t)$ is $\mathit{Stream}^\uparrow \to \bigcup_{i=1}^{n^\uparrow} T_i^\uparrow$. It varies from the type of μ in three points:

1. It does not require a memory configuration. While the monitor maintains its internal memory representation, it is not necessary to reveal details about this for proving functional correctness.

2. The third argument to the memory access function is a natural number representing an offset used to access old values of streams. The semantics requires this information to resolve temporal dependencies such as lookup expressions with offsets. For a monitor, only the most recent outputs are of concern.

3. The output of the monitor does not contain information about input stream values because these are part of the event and do not need to be computed.

The next chapter develops techniques that allow for monitoring a specification with a bounded amount of memory independent of the input sequence and termination time.

Chapter 5

Efficient Monitoring of RTLola

The formal semantics of an RTLola specification starts with an entirely invalid memory and gradually increases the amount of valid information stored in its memory configuration. Since information is never discarded, a naïve realization of the semantics stores all events and computed output values. While this properly reflects the semantics, it is infeasible in practice because its memory consumption grows linearly in the number of streams and the length of the trace. Though the size of the formula is constant at the time the monitor is started, the length of the trace is often unknown and thus unbounded. While this poses little to no problem to state of the art general purpose computers, embedded devices are a different story entirely. They are often subject to strict limitations in terms of weight, space, cost, and energy consumption. Consider, for example, a spacecraft. Every additional gram of payload increases the fuel consumption, resulting in the necessity to bring more fuel, increasing the total weight of the system and so on. Thus, every additional kilobyte of memory can significantly increase the cost of the operation.

Thus, this section presents a procedure to identify statically when values lose their relevancy for the evaluation process and can thus be discarded. Similarly, we identify cases where data can be aggregated without loss of information. Further, we show how to compute an upper bound on the amount of memory needed to evaluate a given RTLola specification.

A closer inspection of the formal semantics reveals that the evaluation process refers back to old memory configuration in three places.

1. When a stream is inactive, it retains its value from the former layer, or — in the 0^{th} layer — from the memory configuration after the preceding relevant timestamp. This, however, does not lead to an increase amount of memory because appropriate bookkeeping allows us to forgo re-duplicating the value.

2. A stream lookup such as $Sync(s^{-}_{j}, n, d)$ accesses the $(n+1)^{\text{st}}$-to-latest value. For instance, `a.offset(by: -1)` becomes the AST node $Offset(a, 1)$ with a positive offset and refers to the second to latest value of `a`; the synchronous lookup `a` refers to the latest one. Offset expressions always include a discrete a-priori determined offset that can be utilized to save memory.

3. In contrast to that, sliding window aggregations are more involved. There is a potentially infinite number of values that are subject to the aggregation because the target stream of a window can have a variable frequency. Yet, the type system and some restrictions on the aggregation functions allow us to bound the memory requirement for sliding windows statically.

In the following, we will show that every value that is further in the past than any discrete set referring to it can be safely discarded. Moreover, utilizing the type system and some restriction on the aggregation functions reduces the memory consumption of sliding windows expressions to a reasonable level; even without assumption on the input frequency. For both of these transformations we show correctness by referring back to the semantics of RTLola. Lastly, we will introduce an algorithm for the evaluation of a specification. The resulting monitor is provably semantically valid.

5.1. Offset Handling

The first source of memory requirements is offsets in stream lookups. All of them are finite and statically available, they do not depend on dynamic data. Thus, it is possible to compute the *storage requirement* of each stream based on the dependency graph. If the storage requirement of s is n, the evaluation potentially needs access to the latest $n+1$ values of s. Triggers generally do not impose any memory requirement because other streams cannot depend on them, so computed values can be discarded immediately. Their storage requirement is therefore 0.

Example 5.1.1. Consider the following specification.

```
input a: Int8
output b: Int8 @5Hz := a.hold().defaults(to: -2)
output c := multiply(a.offset(by: -1).defaults(to: 0), a)
output d := c.offset(by: -1).defaults(to: 1)
```

Stream `b` accesses `a` synchronously over a sample and hold operation. Since there is no offset involved, this only requires to store the latest value of `b`. As opposed to that, `c` accesses `a` with offset 0 and 1. As a result, the storage requirement of `a` is at least 2. Lastly, `d` accessing `c` with offset 1 imposes a storage requirement of at least 2 on `c` and does not influence the requirement of `a` transitively. △

Definition 29 (Storage Requirement)

Def. Storage Requirement

The *storage requirement* $\kappa(s)$ of stream s states how many values of s need to be stored

for the evaluation process. It is the maximum offset of stream lookups with target s computed based on the dependency graph $DG = (V, E)$ (Definition 8). → Sec. 4.2, Page 21

$$\kappa(s_i^-) := \max\{w \mid \exists s_j^\uparrow \colon (s_j^\uparrow, w, s_i^-) \in E \wedge w \in \mathbb{N}\} + 1$$

Note that this definition deliberately excludes window dependencies because these are handled separately as explained in Section 5.2. → Sec. 5.2, Page 57

This information allows for defining a memory representation that contains enough information to be equivalent to the formal semantics and requires only finite memory.

Definition 30 (Finite Memory Configuration)

The finite memory configuration $\overline{M}$ reserves space for each stream based on its storage requirement in a two-dimensional matrix. It further contains information about the *age* of a stream, i.e., the number of values a stream already produced. Stream Age

$$\overline{M} \in (T_1^{\downarrow\kappa(s_1^\downarrow)} \times \mathbb{N}) \times \cdots \times (T_{n^\downarrow}^{\downarrow\ \kappa(s_{n^\downarrow}^\downarrow)} \times \mathbb{N}) \times (T_1^{\uparrow\kappa(s_0^\uparrow)} \times \mathbb{N}) \times \cdots \times (T_{n^\uparrow}^{\uparrow\ \kappa(s_{n^\uparrow}^\uparrow)} \times \mathbb{N})$$

The age is then:

$$age(\overline{M}, s) := \begin{cases} \overline{M}[i][\kappa(s_i^\downarrow) + 1] & \text{if } s = s_i^\downarrow \\ \overline{M}[n^\downarrow + i][\kappa(s_i^\downarrow) + 1] & \text{if } s = s_i^\uparrow \end{cases}$$

The memory access selects the row vector corresponding to the stream based on whether it is an input or an output. Within the row, the access function selects the entry based on the age and absolute offset. The values within the row are ordered latest to oldest, so if the absolute offset corresponds to the age of the stream, the left-most value is selected.

$$\mu(\overline{M})(s)(x) := \begin{cases} \overline{M}[i][age(\overline{M}, s_i^\downarrow) - x] & \text{if } s = s_i^\downarrow \\ \overline{M}[n^\downarrow + i][age(\overline{M}, s_i^\uparrow) - x] & \text{if } s = s_i^\uparrow \end{cases}$$

This finite memory configuration is an integral part of the *finite monitor* $\overline{\mathcal{M}}$. Its evaluation process behaves similar to the formal semantics in that it progresses with relevant timestamps and evaluation layer by evaluation layer. **Def.** Finite Monitor

Definition 31 (Finite Memory Semantics)

Initially, all values are invalid and all ages are 0:

$$\overline{M}_0[s][x] := \begin{cases} \# & \text{if } x \leq \kappa(s) \\ 0 & \text{if } x = \kappa(s) + 1 \end{cases}$$

Then, for relevant timestamp $i > 0$ and evaluation layer $\lambda = 0$, once again the output streams cannot be active and thus do not change. Also, input streams that are not active do not change. Assume $active(s_j^{\downarrow}, i, 0)$.

$$\overline{M}_i^0[j][x] \coloneqq \begin{cases} e_q.data[j] & \text{if } x = 0 \\ \overline{M}_{i-1}[j][x+1] & \text{if } 0 < x \leq \kappa(s_j^{\downarrow}) \\ age(\overline{M}_{i-1}, s_j^{\downarrow}) + 1 & \text{if } x = \kappa(s_j^{\downarrow}) + 1 \end{cases}$$

For a layer $\lambda > 0$, the inputs do not change, and so do inactive outputs. For active output $s_k^{\uparrow}$, the entry $j = k + n^{\downarrow}$ changes as follows:

$$\overline{M}_i^{\lambda}[j][x] \coloneqq \begin{cases} eval(\overline{M}_i^{\lambda-1})(s_{j-n^{\downarrow}}^{\uparrow}.expr) & \text{if } x = 0 \\ \overline{M}_i^{\lambda-1}[j][x+1] & \text{if } 0 < x \leq \kappa(s_k^{\uparrow}) \\ age(\overline{M}_i^{\lambda-1}, s_k^{\uparrow}) + 1 & \text{if } x = \kappa(s_k^{\uparrow}) + 1 \end{cases}$$

→ Sec. 4.4, Page 42

The stream expression evaluation works exactly as in Definition 27, only μ and M are replaced by $\overline{\mu}$ and $\overline{M}$.

Theorem 13 (Correctness of Finite Memory Accesses)**.** *Assume the specification does not contain a sliding window expression. The memory access function μ behaves equivalently for an infinite memory configuration M and a finite one ($\overline{M}$) when both are built according to the rules declared in Definition 26 and Definition 31, respectively. More formally, let $x \leq \kappa(s)$ and let $\hat{t}_i$ be a relevant timestamp, $\lambda \leq \lambda^*$ be any evaluation layer, and $s \in Stream$ be a stream. Then:*

→ Sec. 4.4, Page 41

$$\mu(M_i^{\lambda})(s)(x) = \mu(\overline{M}_i^{\lambda})(s)(x)$$

Note that the exclusion of sliding windows is a heavy restriction on the specification. In the next section, we will introduce a mechanism to remove it. The reason for the restriction becomes evident when stating and proving two lemmas. The first one shows the relation between the *fempty* function and a stream's age. The second shows that the expression evaluation never accesses values without the bounds of $\overline{M}$.

Lemma 14 (Connection of First Empty and Stream Age)**.** *The fempty function always yields the same value as age.*

$$fempty(M_i^{\lambda}, s) = age(\overline{M}_i^{\lambda}, s)$$

Proof.

$$
\begin{aligned}
\mathit{fempty}(M_i^\lambda, s) &= \min\{x \mid \mu(M_i^\lambda)(s)(x) \neq \#\} \\
&= \left(\sum_{k=1}^{i-1} \mathbb{1}_{\exists \lambda\colon \mathit{active}(s,k,\lambda)}\right) + \mathbb{1}_{\exists \lambda' < \lambda\colon \mathit{active}(s,i,\lambda')} \\
&= \left(\sum_{k=1}^{i-1} \mathbb{1}_{\mu(\overline{M}_{k-1})(s)(\kappa(s)) \neq \mu(\overline{M}_k)(s)(\kappa(s))}\right) + \mathbb{1}_{\exists \lambda' < \lambda\colon \mathit{active}(s,i,\lambda')} \\
&= \left(\sum_{k=1}^{i-1} \mathbb{1}_{\mu(\overline{M}_{k-1})(s)(\kappa(s)) = \mu(\overline{M}_k)(s)(\kappa(s))-1}\right) + \mathbb{1}_{\exists \lambda' < \lambda\colon \mathit{active}(s,i,\lambda')} \\
&\qquad\qquad \text{(Definition 31)} \quad \text{→ Sec. 5.1, Page 51} \\
&= \left(\sum_{k=1}^{i-1} \mathbb{1}_{\mathit{age}(\overline{M}_{k-1},s) = \mathit{age}(\overline{M}_k,s)-1}\right) + \mathbb{1}_{\exists \lambda' < \lambda\colon \mathit{active}(s,i,\lambda')} \qquad \text{(Definition 30)} \quad \text{→ Sec. 5.1, Page 51} \\
&= \mathit{age}(\overline{M}_{i-1}^\lambda, s) + \mathbb{1}_{\exists \lambda' < \lambda\colon \mathit{active}(s,i,\lambda')} \\
&= \mathit{age}(\overline{M}_i^\lambda, s)
\end{aligned}
$$

□

The next lemma attempts to correlate the evaluation of stream expressions for regular and finite memory.

Lemma 15 (Attempt: Stream Expression Equivalence with Finite Memory)**.** *The evaluation of an expression via eval, which uses μ, is equivalent for the regular and the finite semantics. For this, assume $\mu(M)(s)(x) = \mu(\overline{\mu})(s)(x)$ for all $s \in \mathit{Stream}$ and $x \leq \kappa(s)$. Then:*

$$\mathit{eval}(M)(s^\uparrow.\mathit{expr}) = \mathit{eval}(\overline{M})(s^\uparrow.\mathit{expr})$$

Upon closer inspection of Definition 27 one can see that there are only two expressions for which the evaluation can diverge: sliding windows and offsets. For the latter, the lemma cannot hold: it requires to access a statically unbounded amount of values. Thus, we will prove a weaker version of the lemma and fix the problem in the next section (Section 5.2). → Sec. 4.4, Page 42 → Sec. 5.2, Page 57

Lemma 16 (Stream Expression Equivalence with Finite Memory)**.** *The evaluation of an expression via eval, which uses μ, is equivalent for the regular and the finite semantics. For this, assume $\mu(M)(s)(x) = \mu(\overline{\mu})(s)(x)$ for all $s \in \mathit{Stream}$ and $x \leq \kappa(s)$. Further, assume $s^\uparrow.\mathit{expr}$ does not contain any sliding window expressions.*

$$\mathit{eval}(M)(s^\uparrow.\mathit{expr}) = \mathit{eval}(\overline{M})(s^\uparrow.\mathit{expr})$$

Proof. The only remaining expression kind that can prove problematic is an offset expression. Assume stream $s^{\uparrow}_j$ accesses s^{-}_k with offset n. The evaluation is defined as:

$$eval_{s^{\uparrow}_j}(M)(Offset(s^{-}_k)) = \begin{cases} \mu(M)(s^{-}_k(x)) & \text{if } x = r2a(M, s^{-}_k, i, Layer(s^{\uparrow}_j), n) \wedge x \geq 0 \\ \# & \text{otherwise} \end{cases}$$

We are only interested in the case accessing M. The offset x is defined in Equation 4.10
→ Sec. 4.4, Page 42
as:

$$\begin{aligned} r2a(M, s^{-}_k, i, \lambda) &= fempty(M, s^{-}_k) - 1 - n + \mathbb{1}_{\exists \lambda' \in [\lambda, \lambda^*]\colon active(s^{-}_k, i, \lambda')} \\ &= age(\overline{M}, s^{-}_k) - 1 - n + \mathbb{1}_{\exists \lambda' \in [\lambda, \lambda^*]\colon active(s^{-}_k, i, \lambda')} \end{aligned} \tag{5.1}$$

→ Sec. 5.1, Page 52
This follows from Lemma 14.

Further, we know:

$$\begin{aligned} \mu(\overline{M})(s^{-}_k)(x) &= \overline{M}[n^{\downarrow} + k][age(\overline{M}, s^{-}_k) - x] \\ &= \overline{M}[n^{\downarrow} + k][age(\overline{M}, s^{-}_k) - (age(\overline{M}, s^{-}_k) - 1 - n + \mathbb{1}_{\exists \lambda' \in [\lambda, \lambda^*]\colon active(s^{-}_k, i, \lambda')})] \\ &\qquad \text{(By Equation 5.1)} \\ &= \overline{M}[n^{\downarrow} + k][age(\overline{M}, s^{-}_k) - age(\overline{M}, s^{-}_k) + 1 + n - \mathbb{1}_{\exists \lambda' \in [\lambda, \lambda^*]\colon active(s^{-}_k, i, \lambda')}] \\ &= \overline{M}[n^{\downarrow} + k][1 + n - \mathbb{1}_{\exists \lambda' \in [\lambda, \lambda^*]\colon active(s^{-}_k, i, \lambda')}] \end{aligned}$$

For the lemma to hold, it suffices to show that $1 + n - \mathbb{1}_{\exists \lambda' \in [\lambda, \lambda^*]\colon active(s^{-}_k, i, \lambda')}$ is no less than 0 and no greater than $\kappa(s^{-}_k)$. The lower bound can be approximated as follows:

$$1 + n - \mathbb{1}_{\exists \lambda' \in [\lambda, \lambda^*]\colon active(s^{-}_k, i, \lambda')} \geq 1 + 0 - \mathbb{1}_{\exists \lambda' \in [\lambda, \lambda^*]\colon active(s^{-}_k, i, \lambda')} \geq 1 + 0 - 1 = 0$$

The upper bound makes use of the definition of $\kappa(s^{-}_k)$ which is always strictly greater than the greatest offset accessing s^{-}_k.

$$\begin{aligned} & 1 + n - \mathbb{1}_{\exists \lambda' \in [\lambda, \lambda^*]\colon active(s^{-}_k, i, \lambda')} \\ \leq\ & 1 + \kappa(s^{-}_k) - 1 - \mathbb{1}_{\exists \lambda' \in [\lambda, \lambda^*]\colon active(s^{-}_k, i, \lambda')} \\ \leq\ & 1 + \kappa(s^{-}_k) - 1 - 0 \\ \leq\ & \kappa(s^{-}_k) \end{aligned}$$

Therefore, all accesses are valid, concluding the proof of the lemma. □

→ Sec. 5.1, Page 52
These two lemmas allow us to prove Theorem 13.

Proof of Theorem 13. Proof by induction on the relevant timestamp, i.e., on $i \in \mathbb{N}$.
1) Induction base: $i = 0$.

Recall that $x \leq \kappa(s_k^-)$. Thus:

$$\mu(M_0^\lambda)(s)(x) = \# = \mu(\overline{M}_0^\lambda)(s)(x)$$

2) Induction hypothesis.

For all cycles up to an arbitrary i, memory accesses on the regular and the finite memory configuration behave equivalently, i.e.:

$$\mu(M_i^\lambda)(s)(x) = \mu(\overline{M}_i^\lambda)(s)(x) \tag{5.2}$$

3) Induction step: $i \rightarrow i+1$.

The induction step requires a nested induction on the evaluation layer $\lambda \in \{0, \ldots, \lambda^*\}$.

3.1) Induction base: $\lambda = 0$.

The lowest layer indicates that changes only occur for input streams. This is true for both the regular and the finite semantics. Recall the definition of the regular semantics (Definition 26) for input stream $s_j^\downarrow$. → Sec. 4.4, Page 41

$$\mu(M_{i+1}^0)(s_j^\downarrow)(x) = \begin{cases} e_q.data[i] & \text{if } x = fempty(M_i^0, s_j^\downarrow) \\ \mu(M_i^0)(s_j^\downarrow)(x) & \text{otherwise} \end{cases}$$

$$= \begin{cases} \overline{M}_{i+1}^0[j][0] & \text{if } x = fempty(M_i^0, s_j^\downarrow) \\ \mu(M_i^0)(s_j^\downarrow)(x) & \text{otherwise} \end{cases} \quad \text{(By Definition 31)} \quad \text{→ Sec. 5.1, Page 51}$$

$$= \begin{cases} \mu(\overline{M}_i^0)(s_j^\downarrow)(age(\overline{M}_i^0, s_j^\downarrow)) & \text{if } x = fempty(M_i^0, s_j^\downarrow) \\ \mu(M_i^0)(s_j^\downarrow)(x) & \text{otherwise} \end{cases}$$

(By Definition 30) → Sec. 5.1, Page 51

$$= \begin{cases} \mu(\overline{M}_i^0)(s_j^\downarrow)(x) & \text{if } x = fempty(M_i^0, s_j^\downarrow) \\ \mu(M_i^0)(s_j^\downarrow)(x) & \text{otherwise} \end{cases} \quad \text{(By Lemma 14)} \quad \text{→ Sec. 5.1, Page 52}$$

$$= \begin{cases} \mu(\overline{M}_i^0)(s_j^\downarrow)(x) & \text{if } x = fempty(M_i^0, s_j^\downarrow) \\ \mu(\overline{M}_i^0)(s_j^\downarrow)(x) & \text{otherwise} \end{cases} \quad \text{(By Equation 5.2)} \quad \text{→ Sec. 5.1, Page 55}$$

$$= \mu(\overline{M}_i^0)(s_j^\downarrow)(x)$$

3.2) Induction hypothesis.

For a fixed cycle $i+1$ and up to an arbitrary evaluation layer λ, memory accesses on the regular and the finite memory configuration behave equivalently, i.e.:

$$\mu(M_{i+1}^\lambda)(s)(x) = \mu(\overline{M}_{i+1}^\lambda(s)(x) \tag{5.3}$$

3.3) Induction step: $\lambda \to \lambda + 1$.

In both the regular and the finite semantics, only output streams can be active in a non-zero layer. Consider output stream $s^{\uparrow}_j$.

$$\mu(M^{\lambda+1}_{i+1})(s^{\uparrow}_j)(x) = \begin{cases} eval_{s^{\uparrow}_j}(M^{\lambda}_{i+1})(s^{\uparrow}_j.expr) & \text{if } x = fempty(M^{\lambda}_{i+1}, s^{\downarrow}_j) \\ \mu(M^{\lambda}_{i+1})(s^{\downarrow}_j)(x) & \text{otherwise} \end{cases}$$

$$= \begin{cases} eval_{s^{\uparrow}_j}(M^{\lambda}_{i+1})(s^{\uparrow}_j.expr) & \text{if } x = fempty(M^{\lambda}_{i+1}, s^{\downarrow}_j) \\ \mu(M^{\lambda}_{i+1})(s^{\downarrow}_j)(x) & \text{otherwise} \end{cases} \quad \text{(By Lemma 16)}$$

→ Sec. 5.1, Page 53

$$= \begin{cases} \overline{M}^{\lambda+1}_{i+1}[j][0] & \text{if } x = fempty(M^{\lambda}_{i+1}, s^{\downarrow}_j) \\ \mu(M^{\lambda}_{i+1})(s^{\downarrow}_j)(x) & \text{otherwise} \end{cases} \quad \text{(By Definition 31)}$$

→ Sec. 5.1, Page 51

$$= \begin{cases} \mu(\overline{M}^{\lambda}_{i+1})(s^{\downarrow}_j)(age(\overline{M}^{\lambda}_{i+1}, s^{\uparrow}_j)) & \text{if } x = fempty(M^{\lambda}_{i+1}, s^{\downarrow}_j) \\ \mu(M^{\lambda}_{i+1})(s^{\downarrow}_j)(x) & \text{otherwise} \end{cases}$$

(By Definition 30)

→ Sec. 5.1, Page 51

$$= \begin{cases} \mu(\overline{M}^{\lambda}_{i+1})(s^{\downarrow}_j)(x) & \text{if } x = fempty(M^{\lambda}_{i+1}, s^{\downarrow}_j) \\ \mu(M^{\lambda}_{i+1})(s^{\downarrow}_j)(x) & \text{otherwise} \end{cases} \quad \text{(By Lemma 14)}$$

→ Sec. 5.1, Page 52

$$= \begin{cases} \mu(\overline{M}^{\lambda}_{i+1})(s^{\downarrow}_j)(x) & \text{if } x = fempty(M^{\lambda}_{i+1}, s^{\downarrow}_j) \\ \mu(\overline{M}^{\lambda}_{i+1})(s^{\downarrow}_j)(x) & \text{otherwise} \end{cases} \quad \text{(By Equation 5.3)}$$

→ Sec. 5.1, Page 55

$$= \mu(\overline{M}^{\lambda}_{i+1})(s^{\downarrow}_j)(x)$$

This concludes both induction steps and therefore the proof. □

We would like to conclude that the finite memory representation is indeed finite. However, so far, the name turns out to be a misnomer. Recall the type of $\overline{M}$:

$$\overline{M} \in (T_1^{\downarrow \kappa(s^{\downarrow}_1)} \times \mathbb{N}) \times \cdots \times (T_{n^{\downarrow}}^{\downarrow \kappa(s^{\downarrow}_{n^{\downarrow}})} \times \mathbb{N}) \times (T_1^{\uparrow \kappa(s^{\uparrow}_0)} \times \mathbb{N}) \times \cdots \times (T_{n^{\uparrow}}^{\uparrow \kappa(s^{\uparrow}_{n^{\uparrow}})} \times \mathbb{N})$$

The natural numbers represent the age of a stream, which is bounded by the length of the run of the monitor, which in turn is unbounded. However, the age is not actually required — it is a *ghost register* only necessary for the correctness proof.

Corollary 17. *The correct computation of an* RTLola *specification without sliding windows requires only a finite amount of memory; the amount can be computed statically.*

Proof. An actual implementation accesses the memory configuration by offset, which is 0 for synchronous accesses and n of $Offset(s^{-}_j, n)$. Assume the accesser is $s^{\uparrow}_k$ and w.l.o.g.

the accessee is an input stream $s_j^- = s_j^\downarrow$. A closer inspection of the memory access of the finite memory configuration when evaluating the offset reveals the following:

$$\begin{aligned}
\mu(\overline{M})(s_j^-)(x) &= \overline{M}[j][age(\overline{M}, s_j^\downarrow) - x] && \text{(Definition 30)} && \text{→ Sec. 5.1, Page 51}\\
&= \overline{M}[j][age(\overline{M}, s_j^\downarrow) - r2a(M, s_j^\downarrow, i, Layer(s_k^\uparrow), n)] && \text{(Definition 27)} && \text{→ Sec. 4.4, Page 42}\\
&= \overline{M}[j][age(\overline{M}, s_j^\downarrow) - (fempty(M, s_j^-) - 1 - n + \mathbb{1})] && \text{(Equation 4.10)} && \text{→ Sec. 4.4, Page 42}\\
&= \overline{M}[j][age(\overline{M}, s_j^\downarrow) - (age(\overline{M}, s_j^-) - 1 - n + \mathbb{1})] && \text{(Lemma 14)} && \text{→ Sec. 5.1, Page 52}\\
&= \overline{M}[j][1 + n - \mathbb{1}]
\end{aligned}$$

Note that the actual indicator condition is irrelevant for the proof and thus omitted. Evidently, both the age and the absolute offset are irrelevant for the evaluation; the only information necessary is the offset and the result of the indicator function. The former is a specification constant. The latter checks whether the accessed stream was active in the current evaluation phase. This is a single bit of information.

Thus, the natural number in the finite memory configuration can be replaced by a single bit for each stream; rendering the finite memory configuration actually finite.

$$|\overline{M}| = \left(\sum_{i \le n^\downarrow} T_i^\downarrow \cdot \kappa(s_i^\downarrow) + 1\right) + \left(\sum_{i \le n^\uparrow} T_i^\uparrow \cdot \kappa(s_i^\uparrow) + 1\right)$$

□

5.2. Sliding Window Handling

A sliding window expression aggregates all values that occurred within a certain time frame. This allows for specifying interesting real-time properties such as

- How many articles were sold over the online shop in the last thirty seconds?
- Is our network target of a denial of service attack manifesting by a rapid increase in incoming connections?
- Did the aircraft cover at least 15m per minute according to the GPS sensor data during the entire mission?
- Is there a particular peak time where the server's response time exceeds a threshold?

The expressiveness, however, comes with a computational overhead and hefty impact on the memory requirements. Computing a sliding window requires to keep track of the arrival times of all data points that are subject to the aggregation. As soon as a value is older than the specified window duration, it needs to be evicted so it is no longer

taken into account. In an asynchronous model such as the one covered in this thesis, there is no bound on the number of values occurring within this time frame. Ergo, the required amount of memory is already unbounded merely to decide which values are relevant for the window. Even if this problem were solved, some aggregations like the median requires information about all data points to be computed.

This section presents a remedy for the worst case memory consumption by identifying "nice" aggregations with a finite memory requirement and making use of periodic stream computations to compute aggregations faster and with less memory.

5.2.1. Properties of Aggregation Functions

The most naïve implementation of a sliding window stores the entire input sequence $v_1,\dots,v_n$ and aggregates all qualifying values, e.g. $\gamma(v_k,\dots,v_n)$. While this is a sound approach, it requires to re-aggregate the entire sequence as soon as a new value arrives even though the sequences only differ in a single value. The same problem occurs when computing a running aggregation, i.e., computing a sequence of outputs $r_1,\dots,r_n$ with $r_k = \gamma(v_1,\dots,v_k)$ for any $k \leq n$. For many commonly used aggregation functions, there are efficient algorithms for the running aggregations.

As an example, consider the *median*, i.e. the element v_i where exactly half of the element in $v_1,\dots,v_n$ are greater and half are less than v_i. Using an algorithm often referred to as "median of medians" [60], the median can be computed in $\mathcal{O}(n)$. However, using this method, the running median is in $\mathcal{O}(n^2)$ because the intermediate results of the median of medians computation are not easily reusable. However, when storing all received values in a sorted data structure and just maintaining the order for each new one, the running time decreases. Since insertion is possible in $\mathcal{O}(\log_2(n))$, and n elements need to be inserted, the overall running time amounts to $\mathcal{O}(n\log_2(n))$.

Yet, this method neither has constant running time for each new value, nor constant memory overhead. This can be solved with stronger requirement on the aggregation function. An aggregation that is a *list homomorphism* allows for re-usable intermediate aggregation results, resulting in a constant running time and constant memory requirements in the length of the input sequence.

Definition 32 (Homomorphism [61])

Def. List Homomorphism

A *list homomorphism* $\gamma\colon A^* \to B$ can be split into four components:

- an unary function $map_\gamma\colon A \to T$ lifting a single value into an intermediate representation
- a unary finalization function $fin_\gamma\colon T \to B$ lowering an intermediate value to a result
- an associative binary reduction function $\circ_\gamma\colon T\times T \to T$, i.e., $(a\circ_\gamma b)\circ_\gamma c = a\circ_\gamma(b\circ_\gamma c)$
- a neutral element $\varepsilon_\gamma \in T$ w.r.t. $\circ_\gamma$, i.e. $a \circ_\gamma \varepsilon_\gamma = \varepsilon_\gamma \circ_\gamma a = a$ for any $a \in T$.

The combination of these operations is equivalent to the immediate aggregation for any sequence $(v_1, \ldots, v_n) \in A^n$.

Theorem 18 (Meertens [61])**.** *The aggregation of $v_1, \ldots v_n$ using a list homomorphism γ can be broken into arbitrary sub-aggregations. Let $(I_i)_{i \leq k} = ((x_{i,j})_{j \leq |I_i|})_{i \leq k}$ for some $k \in \mathbb{N}$ be an ordered partition (see Definition 6) of the interval $[1, \ldots, n]$.* → *Sec. 3.2, Page 13*

$$\gamma(v_1, \ldots, v_n) = fin_\gamma((map_\gamma(x_{1,1}) \circ_\gamma \ldots \circ_\gamma x_{1,|I_1|}) \circ_\gamma \ldots \circ_\gamma (map_\gamma(x_{k,1}) \circ_\gamma \ldots \circ_\gamma map_\gamma(x_{k,|I_1|})))$$

This allows for arbitrary sub-aggregation of the input sequence. Moreover, when computing the running aggregation, intermediate results can be stored. Suppose the k^{th} output is $r_k = fin_\gamma(map_\gamma(v_1) \circ_\gamma \ldots \circ_\gamma map_\gamma(v_k))$. The intermediate result i_k used as input for fin_γ can be re-used when a new value v_{k+1} arrives, so $r_{k+1} = fin_\gamma(i_k \circ_\gamma map_\gamma(v_{k+1}))$. Consequently, the memory consumption is bounded by the constant size of type T.

Example 5.2.1. Many common aggregation functions are list homomorphisms:

Average with $map_{avg}(v) = (v, 1)$, $fin_{avg}((s, c)) = \frac{s}{c}$, $(s_1, c_1) \circ_{avg} (s_2, c_2) = (s_1 + s_2, c_1 + c_2)$, and $\varepsilon_{avg} = (0, 0)$. Here, the intermediate value sums all input values up and counts them. The devision of both values yields the average.

Extrema with $map_{ext}(v) = v$, $fin_{ext}(v) = v$, $a \circ_{max} b = \max(a, b)$ for maximization or $a \circ_{min} b = \min(a, b)$ for minimization, and $\varepsilon_{max} = -\infty$ or $\varepsilon_{min} = \infty$.

Integration where the input values are tuples $(v, t) \in A \times \mathbb{R}_+$ of a value and its arrival timestamp. The integration uses a trapezoid abstraction, illustrated in Figure 5.1, where each intermediate value represents a section of the graph consisting the left-most value and timestamp, the right-most value and timestamp, as well as the overall volume. The neutral element ε_γ is a special $\perp$ value with no semantics other than being the neutral element. Then, $map_\int((p, t)) = (p, t, p, t, 0)$, $fin_\int(\perp) = 0$ of the neutral element and $fin_\int((p^L, t^l, p^R, t^R, V) = V$, and lastly

$$(p_1^L, t_1^L, p_1^R, t_1^R, V_1) \circ_\int (p_2^L, t_2^L, p_2^R, t_2^R, V_2) = (p_1^L, t_1^L, p_2^R, t_2^R, \frac{1}{2}(p_1^R + p_2^L)(t_2^L - t_1^R) + V_1 + V_2)$$

If either operand is $\perp$, $\circ_\int$ yields the other operand, if both are $\perp$, the result is $\perp$ as well. Note that the binary reduction is associative and not commutative.

Variance is a statistical measure for the spread of a data set compared to their mean value. For a sequence $(x_i)_{i \leq n}$ resulting from a random variable X with mean μ it is defined as:

$$\text{Var}(X) := \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2$$

A naïve implementation requires two passes: the first computes the mean, the second computes the sum of squares of differences. In 1962, Welford [62] found

an iterative algorithm requiring only a single pass. This is a special case of the parallel algorithm proposed by Chen et al. [63]. They split in the input sequence around some $1 \leq k \leq n$ and compute the count, mean, and variance of each sequence separately. Assume n_X is the count, V_X is the Variance, and μ_X is the mean of the lower or upper half of the sequence for $X \in \{L, U\}$. The reduction of the homomorphism is then:

$$L \circ_{\text{Var}} U = (L \circ_{\text{Var}_n} U, L \circ_{\text{Var}_V} U, L \circ_{\text{Var}_\mu} U)$$

with

$$(n_L, V_L, \mu_L) \circ_{\text{Var}_n} (n_U, V_U, \mu_H) = n_L + n_H$$

$$(n_L, V_L, \mu_L) \circ_{\text{Var}_V} (n_U, V_U, \mu_H) = V_L + V_H + (\mu_U - \mu_L)^2 \frac{n_L n_U}{n_L + n_U}$$

$$(n_L, V_L, \mu_L) \circ_{\text{Var}_\mu} (n_U, V_U, \mu_H) = \mu_L + (\mu_U - \mu_L) \frac{n_U}{n_L + n_U}$$

The other components of the homomorphism are $map_{\text{Var}}(x_i) = (1, 0, x_i)$, which is the count, variance and mean for singleton sequences, $fin_{\text{Var}}((n, V, \mu)) = V$ extracts the variance out of the triple, and $\varepsilon_{\text{Var}} = (0, 0, 0)$ as neutral element.

Covariance The covariance of two random variables X and Y is defined as:

$$\text{Cov}(X, Y) := \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu_X)(y_i - \mu_Y)$$

A similar approach is possible for computing the covariance. The idea remains the same, only the reduction of the variance $\circ_{\text{Var}_V}$ needs to be replaced by $\circ_{\text{Cov}_C}$ defined as follows:

$$(n_L, C_L, \mu_L) \circ_{\text{Cov}_C} (n_U, C_U, \mu_H) = \frac{1}{n}(n_L n_U + n_U C_U + n_L(\mu_L - \mu)^2 + n_U(\mu_U - \mu)^2)$$

△

List homomorphisms thus allow us to split the input sequence into arbitrary chunks and pre-aggregate the content thereof into intermediate values before reducing them to the final result. This especially enables an efficient computation of running aggregation.

5.2.2. Memory Reduction by Pre-Aggregation in Periodic Streams

The difference between running aggregations and sliding windows is that a sliding window evicts certain values when their arrival time is sufficiently far in the past. This decision requires access to the timestamps of all values affected by the sliding window. Since there is no limit on the size of the input sequence, the required memory is unbounded.

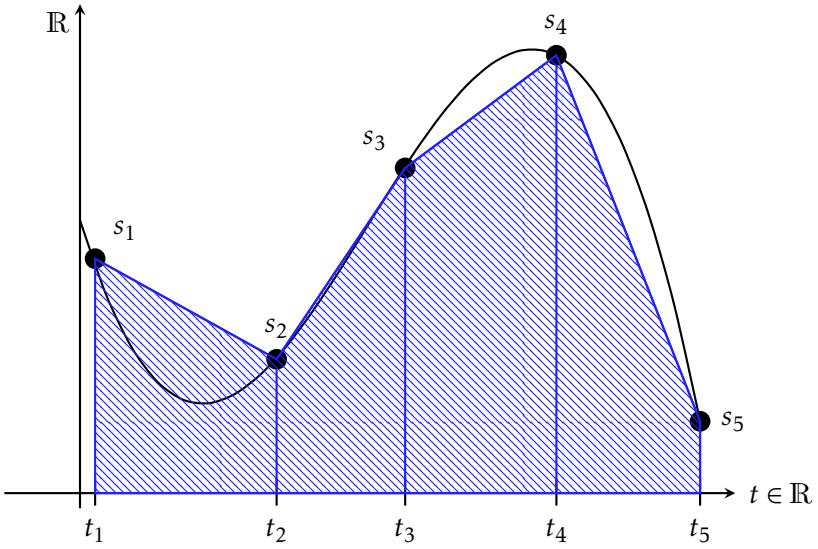


Figure 5.1.: Illustration of the trapezoid abstraction. The underlying function is sampled at time point $(t_i)_{1 \leq i \leq 5}$ with values $(s_i)_{1 \leq i \leq 5}$. The abstraction computes the area of a trapezoid between two samples, which are connected by a first-order hold. Depending on the sampling, the approximation is arbitrarily imprecise (s_1 to s_2) or almost exact (s_2 to s_3).

To get rid of this problem, recall the type inference rules for sliding window expressions in Equation 4.7: it requires a periodic pacing type. This affects the encompassing output stream, requiring it to be a periodic stream. These are *isochron*, so the points in time when the stream will be evaluated are know a priori. This entails that each point of the real time axis can be mapped to a set of sliding windows to which they contribute. This can be seen in Figure 5.2: In the example window, each value corresponds to exactly two sliding window evaluations. The computation time of each of the two evaluations is a priori fixed due to the isochronicity. As a result, the precise arrival time of a value becomes irrelevant: if a sliding window of length 1s is evaluated with 1Hz, a value arriving at $t_1 = 0.34$s and a value arriving at $t_2 = 0.61$s behave equivalently in terms of eviction. Both are relevant up until $t = 1$s, inclusively, and irrelevant afterwards because there will not be another evaluation until $t = 2$s. Note that the arrival order can still have an impact on the aggregation functions when its reduction function is not commutative such as $\circ_{\int}$.

→ Sec. 4.3, Page 34
Isochronicity

To formalize and prove this observation, recall the semantics of the sliding window expression from Definition 27. It computes the number of relevant values using *inwin* and aggregates the affected values.

→ Sec. 4.4, Page 42

We will divide the real timeline into a finite number of equal-sized chunks based on the evaluation frequency of the window. The values occurring within one chunk

are pre-aggregated and only the intermediate values are stored. When evaluating the sliding window, all intermediate values are reduced and the result finalized.

Consider the window $w = \mathit{Window}(s_j^-, \delta, \gamma)$ for a homomorphism $\gamma\colon A^* \to B$ with mapping $\mathit{map}_\gamma\colon A \to T$ and $\pi, \mathit{lift}(B) \models w$ for some frequency π. The number of chunks chc_w and the time each chunk represents chd_w is then:

$$\mathit{chc}_w := \frac{\operatorname{lcm}(\delta, \pi^{-1})}{\pi^{-1}} \qquad \mathit{chd}_w := \gcd(\delta, \pi^{-1})$$

Proposition 19. *The set of chunks cover the entire duration of the window:*

$$\mathit{chc}_w \cdot \mathit{chd}_w = \delta$$

Proof. Note that $\gcd(a,b)\cdot\operatorname{lcm}(a,b) = ab$ for any positive natural numbers a and b. To see that this identity holds, consider the prime number decomposition of a and b.

$$a = \prod_{i=1}^{k_a} p_i^{x_{a,i}} \qquad b = \prod_{i=1}^{k_b} p_i^{x_{b,i}}$$

for natural numbers $k_a, k_b, (k_{a,i})_{l \le k_a}, (k_{b,i})_{i \le k_b}$ and prime numbers $(p_i)_{i \le \max(k_a,k_b)}$. Assume the following:

$$x_{y,i} = 1 \text{ for } i > k_y \text{ and } y \in \{a, b\} \tag{5.4}$$

Then:

$$\begin{aligned}
\gcd(a,b)\operatorname{lcm}(a,b) &= \prod_{i=1}^{\max(k_a,k_b)} p_i^{\max(x_{a,i},x_{b,i})} \prod_{i=1}^{\max(k_a,k_b)} p_i^{\min(x_{a,i},x_{b,i})} \\
&= \prod_{i=1}^{\max(k_a,k_b)} p_i^{\max(x_{a,i},x_{b,i})} p_i^{\min(x_{a,i},x_{b,i})} \\
&= \prod_{i=1}^{\max(k_a,k_b)} p_i^{\max(x_{a,i},x_{b,i})+\min(x_{a,i},x_{b,i})} \\
&= \prod_{i=1}^{\max(k_a,k_b)} p_i^{x_{a,i}+x_{b,i}} \\
&= \prod_{i=1}^{\max(k_a,k_b)} p_i^{x_{a,i}} p_i^{x_{b,i}} \\
&= \left(\prod_{i=1}^{\max(k_a,k_b)} p_i^{x_{a,i}}\right) \cdot \left(\prod_{i=1}^{\max(k_a,k_b)} p_i^{x_{b,i}}\right)
\end{aligned}$$

$$= \left(\prod_{i=1}^{k_a} p_i^{x_{a,i}}\right) \cdot \left(\prod_{i=1}^{k_b} p_i^{x_{b,i}}\right) \qquad \text{(Equation 5.4)}$$

$$= ab$$

Thus:

$$chc_w \cdot chd_w = \frac{\gcd(\delta, \pi^{-1})\operatorname{lcm}(\delta, \pi^{-1})}{\pi^{-1}} = \frac{\delta\pi^{-1}}{\pi^{-1}} = \delta$$

□

We thus reserve *chc* chunks, each representing *chd* seconds of the time axis, see Figure 5.2. Similarly to the memory configuration, let $\Omega\colon \mathcal{W} \to \mathbb{N} \to T$ be a *window configuration*, i.e., additional memory for each window. The first argument is the respective window expression, the second argument identifies the chunk and is undefined for values greater than *chc*. Formerly, for every layer, the memory configuration was updated according to the semantics. After λ^* steps, the last memory configuration became the starting point for the next relevant timestamp. Now, after evaluating a layer, an additional evaluation step concerning the window configuration takes place.

Definition 33 (Window Configuration Semantics)

Def. Window Configuration Semantics

The *window configuration* is updated whenever the target of the window is active in the respective layer. The first chunk of memory always represents the current point in time on the real time axis. This invariant is maintained after each full evaluation cycle. Thus, only the first chunk of memory is ever changed within the cycle. Assume relevant timestamp $\hat{t}_i$.

$$\Omega_i^0(w)(1) := \begin{cases} \Omega_{i-1}(w)(1) \circ_\gamma map_\gamma(\mu(M_i^0)(s_j^-)(r2a(M_i^0, s_j^-, i, 0, 0))) & \text{if } active(s_j^-, i, 0) \\ \Omega_{i-1}(w)(1) & \text{otherwise} \end{cases}$$

For $\lambda > 0$ the only difference is that we copy over the value from the last layer rather than the end of the last phase.

$$\Omega_i^\lambda(w)(1) := \begin{cases} \Omega_{i-1}^{\lambda-1}(w)(1) \circ_\gamma map_\gamma(\mu(M_i^0)(s_j^-)(r2a(M_i^\lambda, s_j^-, i, \lambda, 0))) & \text{if } active(s_j^-, i, \lambda) \\ \Omega_{i-1}^{\lambda-1}(w)(1) & \text{otherwise} \end{cases}$$

All other parts of the memory do not change, so for $k > 1$:

$$\Omega_i^\lambda(w)(k) := \begin{cases} \Omega_i^{\lambda-1}(w)(k) & \text{if } \lambda > 1 \\ \Omega_{i-1}(w)(k) & \text{otherwise} \end{cases}$$

Lastly, if $\hat{t}_i$ is a multiple of π^{-1} it marks the eviction of a chunk. Thus, all chunks get shifted by one to the right. This deletes the content of the oldest chunk. The first one

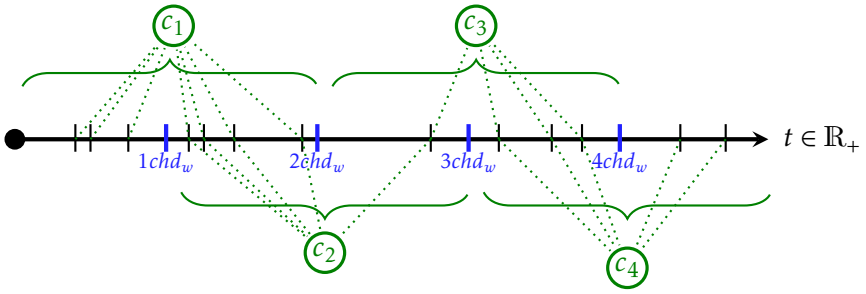


Figure 5.2.: Illustration of a sliding window over the real time axis. The window has type $\pi, \tau \models (s_i^-, 2\pi^{-1}, \gamma)$. Since the duration of the window is twice the period of π^{-1}, each data point of s_i^- is relevant for two window evaluations.

assumes the value of ε_γ, i.e., it gets reset.

$$\Omega_i(w)(k) := \begin{cases} \Omega_i^{\lambda^*}(w)(k) & \text{if } \pi^{-1} \nmid \hat{t}_i \\ \varepsilon_\gamma & \text{if } \pi^{-1} \mid \hat{t}_i \wedge k = 1 \\ \Omega_i^{\lambda^*}(w)(k-1) & \text{if } \pi^{-1} \mid \hat{t}_i \wedge k > 1 \end{cases}$$

As a result, the evaluation needs access to the window configuration as well as the memory configuration.

Definition 34 (Sliding Window Access)

→ Sec. 4.4, Page 42

The evaluation of stream expressions remains as in Definition 27 except for window accesses. Assume stream $s_k^\uparrow$ with evaluation frequency π accesses window w with target s_j^-, duration δ and aggregation γ which is a list homomorphism:

$$eval^*_{s_k^\uparrow}(M, \Omega)(w) := fin_\gamma(\Omega(w)(1) \circ_\gamma \dots \circ_\gamma \Omega(w)(\delta\pi))$$

→ Sec. 4.4, Page 38
→ Sec. 4.4, Page 41

Recall that by Definition 23, every multiple of π^{-1} is a relevant timestamp. Moreover, Definition 26 states that the window is only evaluated at exactly these points in time.

Theorem 20 (Correct Window Access)**.** *Let $\hat{t}_i$ be the i^{th} relevant timestamp with $\pi^{-1} \mid \hat{t}_i$ where pi is the pacing type of $s_k^\uparrow$. Moreover, let Ω be computed as declared in Definition 33 and $w = Window(s_j^-, \delta)$. Then, the memory efficient stream access yields the same value as the semantics of* RTLola *dictate.*

$$eval_{s_k^\uparrow}(M)(w) = eval^*_{s_k^\uparrow}(M, \Omega)(w)$$

→ Sec. 5.2, Page 59

Proof. By Theorem 18 it suffices to show that Ω accounts for the latest $inwin_w(i)$ values of s_j^-. By definition of relevant timestamps, we know that each of these events can be

assigned exactly one relevant timestamp $\hat{t}_\eta$ with $\hat{t}_i - \delta < \hat{t}_\eta < \hat{t}_i$. Ω consists of *chc* chunks, each representing *chd* time units. By Proposition 19, the chunks cover δ time units. So, → Sec. 5.2, Page 62 it remains to be shown that within δ time units, at most *chc* chunks get evicted. By Definition 33, an eviction takes place for every relevant timestamp that is a multiple of π^{-1}, thus we require:

$$\delta \geq chc_w \cdot \pi^{-1} = \frac{\text{lcm}(\delta, \pi^{-1})}{\pi^{-1}} \cdot \pi^{-1} = \text{lcm}(\delta, \pi^{-1}) \cdot \pi \cdot \pi^{-1} = \text{lcm}(\delta, \pi^{-1})$$

This holds by definition of lcm. □

Corollary 21 (Static Memory Bound for Sliding Windows)**.** *For a valid* RTLola *specification with homomorphic aggregations, there is a static bound on the amount of memory required for window configurations.*

Proof. Theorem 20 shows that the window configuration suffices to evaluate homomorphic sliding window aggregations correctly.

The number of sliding windows is a specification constant. The window configuration for each window according to Definition 33 is a function with domain → Sec. 5.2, Page 63 $\{1, \ldots, chc_w\} \subset \mathbb{N}$ for window w. chc_w depends on δ, which is a specification constant, and π, which can be computed prior to starting the monitor in terms of the type analysis. The function maps each of these natural numbers to a value of type T, i.e., the intermediate value of the aggregation function's unary mapping function. Since the size of T is statically known, the window requires $|T| \cdot chc_w$. Ergo, the size of the entire window configuration is:

$$|\Omega| = \sum_{w \in \mathcal{W}} |T_w| \cdot chc_w$$

□

5.2.3. Efficient Eviction of Values

The efficient sliding window computation allows us to reduce the required memory to a finite value with static bound. Yet, it entails a repetition of labor that can be eradicated for some aggregation functions. Assume that the computation of a window with aggregation function γ requires n intermediate values. Consider, for illustration, the infinite sequence of intermediate values $(v_i)_{i \in \mathbb{N}}$ and assume the finalization function coincides with the identity function. The k^{th} and $k+1^{\text{st}}$ computation of the sliding window is thus:

$$\begin{aligned} r_k &= fin_\gamma(map_\gamma(v_{k-n}) \circ_\gamma \underbrace{map_\gamma(v_{(k+1)-n}) \circ_\gamma \ldots \circ_\gamma map_\gamma(v_k)}) \\ r_{k+1} &= fin_\gamma(\overbrace{map_\gamma(v_{(k+1)-n}) \circ_\gamma \ldots \circ_\gamma map_\gamma(v_k)} \circ_\gamma map_\gamma(v_{k+1})) \end{aligned} \tag{5.5}$$

Evidently, most of the computation is the same, especially for large values of n. The two points of difference are the addition of v_{k+1} for r_{k+1} and the exclusion of v_{k-n} due to eviction. The latter point is problematic: A homomorphism does not necessarily provide a removal operation. Ideally, the binary reduction $\circ_\gamma$ would be invertible, i.e., if $a \circ_\gamma b = c$, we could split c into a and b retroactively. This is an extreme restriction on $\circ_\gamma$ because it requires the function to be injective, which many common functions such as addition are not.

Fortunately, the computation has access to all intermediate values, including the value to be evicted. Thus, full invertibility is not necessary; *left-invertibility* suffices.

Definition 35 (Left-Invertibility)

Def. Left-Invertibility

A binary function $\circ\colon A \times B \to C$ is *left-invertible* if the right argument can be reconstructed based on the result and the first argument.

$$a \circ b = c \implies c \circ^{-1} a = b$$

Example 5.2.2. The average is left-invertible with $(s,c) \circ_{avg}^{-1} (s^L, c^L) = (s - s^L, c - c^L)$, counting, summation and products are trivially left-invertible. Extrema and trapezoid integration, however, are not. △

In the general example above (Equation 5.5), r_{k+1} can thus be computed based on r_k with the following formula:

$$r_{k+1} = (r_k \circ_\gamma^{-1} map_\gamma(v_{k-n})) \circ_\gamma map_\gamma(v_{k+1})$$

More generally:

Theorem 22 (Correctness of Efficient Windows)**.** *Assume stream $s_k^{\uparrow}$ with evaluation frequency π accesses window w with target s_j^-, duration δ, and list-homomorphic aggregation γ, which has a left-invertible reduction function $\circ_\gamma$. The access occurs at $\hat{t}_i = n\pi^{-1}$ for $n > 0$ and $\hat{t}_j = (n-1)\pi^{-1}$ is the last relevant timestamp in which the window was computed. Lastly, assume that $\Omega_j(w)(chc+1)$ contains the cached result of the last window access before finalization, with the evicted value already removed.*

$$\Omega_j(w)(chc+1) = (\Omega_j(w)(1) \circ_\gamma \ldots \circ_\gamma \Omega_j(w)(chc)) \circ_\gamma^{-1} \Omega_j(w)(chc)$$

Then:

$$(\Omega_j(w)(chc+1) \circ_\gamma^{-1} \Omega_j(w)(1)) \circ_\gamma \Omega_i(w)(chc) = eval_{s_k^{\uparrow}}(M, \Omega_i)(w)$$

Proof.

$$(\Omega_j(w)(chc+1) \circ_\gamma^{-1} \Omega_j(w)(1)) \circ_\gamma \Omega_i(w)(chc)$$

$$= ((\Omega_j(w)(1) \circ_\gamma \ldots \circ_\gamma \Omega_j(w)(chc)) \circ_\gamma^{-1} \Omega_j(w)(1)) \circ_\gamma \Omega_i(w)(chc) \quad \text{(Def. } \Omega_j(w)(\delta\pi+1)\text{)}$$

$$= (\Omega_j(w)(2) \circ_\gamma \ldots \circ_\gamma \Omega_j(w)(chc)) \circ_\gamma \Omega_i(w)(chc) \quad \text{(Def. } \circ_\gamma^{-1}\text{)}$$

$$= (\Omega_i(w)(1) \circ_\gamma \ldots \circ_\gamma \Omega_i(w)(chc-1)) \circ_\gamma \Omega_i(w)(chc) \quad \text{(Def. } \Omega\text{)}$$

$$= \Omega_i(w)(1) \circ_\gamma \ldots \circ_\gamma \Omega_i(w)(chc-1) \circ_\gamma \Omega_i(w)(chc) \quad \text{(Asso. } \circ_\gamma\text{)}$$

$$= eval^*_{s_k^\uparrow}(M, \Omega_i)(w) \quad \text{(Definition 34)}$$ →Sec. 5.2, Page 64

$$= eval_{s_k^\uparrow}(M, \Omega_i)(w) \quad \text{(Theorem 20)}$$ →Sec. 5.2, Page 64

□

This technique reduces the computational overhead linearly in the size of *chc* for every cycle in which the window is evaluated. It increases the memory consumption by the size of one intermediate value of γ. This accounts for the memorized last pre-aggregated non-finalized value $\Omega_j(w)(chc+1)$.

In the following, $eval^*$ denotes the efficient window evaluation applying the left-invertibility technique whenever possible.

5.3. Finite Memory Evaluation

This section will unite the last two section. We will incorporate the finite memory computation from Section 5.1 for any expression but sliding windows with the techniques detailed in Section 5.2. The resulting evaluation function agrees with the semantics and is realizable with finite memory. We will present a monitoring algorithm obeying the memory bound.

→Sec. 5.1, Page 50
→Sec. 5.2, Page 57

5.3.1. Finite Memory Semantics

Theorem 23 (Finite Memory Monitoring)**.** *A valid* RTLola *specification can be accurately evaluated with finite memory.*

Proof. Corollary 17 shows that *eval* computes correct results for specifications without sliding windows while requiring only a finite amount of memory. Moreover, Corollary 21 shows that $eval^*$ computes sliding windows correctly with finite memory, utilizing window configurations. In combination, a memory efficient monitor uses $eval^*$ with the finite memory configuration $\overline{M}$. Notice that these two techniques modify disjoint parts of the expression evaluation and are thus seamlessly compatible. □

→Sec. 5.1, Page 56
→Sec. 5.2, Page 65

Chapter 6

RTLola in Practice

6.1. Syntactic Sugar

Syntactic Sugar

Desugarization

In this section, we will introduce a selection of *syntactic sugar*. These constructs are no extension to RTLola as any specification with syntactic sugar be transformed into a valid "vanilla" RTLola specification. This transformation is called *desugarization*. Yet, desugarization and regular treatment is not always desirable: a special treatment may significantly increase overall performance of the monitor.

Example 6.1.1. Consider, for example, the following specification featuring a discrete window operation:

```
input a: Int8
output b: Int32 := a.aggregate(discrete: 1000000000, σ)
```

The output stream sums up the last 10^9 values of the input stream. This is syntactic sugar for a specification that accesses the last 10^9 values of `a` one by one and sums up the results. Each access is surrounded by a default operation with the neural element w.r.t. the aggregation as alternative value.

```
input a: Int64
output b: Int64 :=
  a
  + a.offset(by: -1).defaults(to: 0)
  + a.offset(by: -2).defaults(to: 0)
  ...
  + a.offset(by: -999999999).defaults(to: 0)
```

The benefits of syntactic sugar become immediately clear considering the length of the desugarized specification. Apart from that, the monitor for the desugarized specification needs to perform $10^9 - 1$ addition operations per evaluation step. A special treatment similar to the aggregation described in Section 5.2 cannot reduce the memory consumption but significantly reduces the runtime overhead. Per evaluation cycle, the new value of `a` needs to be added to the memoized sum and the 10^9+1^{st} value needs to be evicted, amounting to a single subtraction. △

→ Sec. 5.2, Page 57

We will now list the most prominent syntactic sugar for RTLola and show the desugarization. Special treatment of these constructs is not always beneficial and remains an implementation detail.

Infix notation The desugarized version in Example 6.1.1 already uses syntactic sugar for the addition: infix notation. Strictly speaking, addition is a binary function that requires prefix notation in "vanilla" RTLola. However, infix notation feels more natural to most programmers and thus a crucial feature. Similarly to many modern[1] object-oriented programming languages like Scala, C++ and Rust, infix notation is reduced to a base form. In Scala, for example, the expression `3+7` is desugarized to `Int(3).+(Int(7))`, where `+` is a polymorphic class function of `Int`.

In general, any binary infix operator $\circ$ in an RTLola specification `e1` $\circ$ `e2` is transformed into $\circ$`(e1, e2)`. Operator precedence applies as usual, i.e., for two left-associative operators $\circ_1$ and $\circ_2$ where $\circ_1$ has precedence over $\circ_2$, the expression `e1` $\circ_1$ `e_2` $\circ_2$ `e3` becomes $\circ_1$`(e1,` $\circ_2$`(e2, e3))`. Appropriate precedence tables and their treatment have been discussed thoroughly in literature, so the interested reader is referred to [64, 65, 66][2].

Remark 6.1.1 (Conditional Expressions)**.** *Note that the infix notation also applies to non-binary operations such as conditional expressions, often referred to as 'if-then-else' expressions. This is a ternary expression with three arguments: the boolean* condition*, the* consequence *and the* alternative*. If the condition is true, the result of the consequence is the result of the conditional expression; otherwise it is the result of the alternative. In imperative programming languages, these expressions often have a lazy evaluation model, i.e., either the consequence* or *the alternative is evaluated, not both. Each sub-expression can lead to side effects when evaluated spuriously, special treatment is required. In* RTLola*, however, there are no side effect as can be seen in the memory semantics in Definition 26. An expression cannot assign values; only the evaluation of an entire stream expression is saved. This functional style prevents side effects, allowing for computing both the consequence and alternative without repercussions.*

→ Sec. 4.4, Page 41

Tuples A tuple is a statically determined list of values. The value types of each entry may differ while the pacing type needs to be the same. Tuples can be constructed with the parenthesis operator. The projection function π extracts single values from the tuple. The desugarized specification transforms the tuple stream into a series of streams. The projection then selects the appropriate stream.

Example 6.1.2. The following specification constructs a tuple stream and accesses the entries separately with the tuple projection.

```
input first: Int8
input second: Bool
output tuple: (Int8, Bool) := (first, second)
```

[1] Hereby, we want to acknowledge any future reader who is smirking about a long-dead language being called "modern".

[2] The publications dates are already an indication that this is not a recent problem.

```
output extract_fst: Int8 := π(c, 1)
output extract_snd: Bool := π(c, 2)
```

Desugarization first transforms the parenthesis operator into a function call, i.e., `output tuple: (Int8, Bool):= create_tuple(first, second)`. Further desugarization then yields:

```
input first: Int8
input second: Bool
output tuple_1: Int8 := first
output tuple_2: Bool := second
output extract_fst: Int8 := tuple_1
output extract_snd: Bool := tuple_2
```

△

More formally, the desugarization consists of several steps, applied successively until no further replacement takes place.

1. Any output stream $s^{\uparrow}_i$ containing the `create_tuple` function is split into m output streams $s^{\uparrow}_{i_\eta}$ for each $\eta \in \{1,\dots,m\}$ where the function expression is replaced by the η^{th} argument to `create_tuple`. Assume $s^{\uparrow}_i$ is has name `a`. The new streams then have name `a_1`, up to `a_m`.

2. Functions of arity n defined on tuples of length $\ell_1,\dots,\ell_n$ are transformed into $(\sum_{i=1}^{n} \ell_i)$-ary functions. The sub-expression for the k^{th} argument is replaced by ℓ_k sub-expressions, each accessing one of the created streams for the tuple.

3. Any stream accessing $s^{\uparrow}_i$ is split in the same fashion $s^{\uparrow}_i$ is split. The η^{th} variant accesses `a_`η.

4. Tuple projection expressions are replaced by sub-expressions accessing the respective split stream.

Elaborate Triggers In vanilla RTLola, triggers are always synchronous lookups targeting an output stream. This restriction can be lifted, allowing for general expressions in a trigger. The concrete syntax thus becomes:

```
trigger e "msg"
```

Here, `e` is any expression of value type *Bool*. The desugarization creates a new output stream $s^{\uparrow}_i$ with stream expression `e` and value type annotation *Bool*. This, however, still limits triggers in that the expression needs to have an event-type. Thus, the concrete syntax also allows for an explicit pacing type annotation:

```
trigger @nHz e "msg"
```

In this case, the annotation is moved to the fresh output stream:

```
output a: Bool @nHz := e
trigger a "msg"
```

Conservative Sliding Windows Sliding windows provide a way to check lower bounds on streams. Consider the following specification:

```
input heart_beat: Bool
output num_beats: Int8 @1Hz := heart_beat(over: 5s, count)
trigger num_beats < 100
```

The input stream `heart_beat` produces a value whenever a heart beat is detected. If the number of heart beats in the last 5s drops below 100, the patient's heart rate is below 20. This alarming situation constitutes a violation of the specification, triggering an alarm. However, even a healthy albeit low resting heart rate of 55 triggers an alarm at the beginning of a monitor execution. This is because `num_beats` is computed every second and all present values of the last 5s get counted. If the number of existing values is only around 50 because the execution started 1s ago, a trigger goes off. This can be prevented if the window is only evaluated after the entire duration of the window has already passed at least once.

A variation of the sliding window operation provides this functionality. Rather than calling the first argument `over`, it is called `over_exactly`. This operation returns an optional value, requiring a default value for the first evaluations of the window when the duration has not passed, yet. It turns out, the variation is also just syntactic sugar. The example above turns into the following desugarized specification for the default value `dft`:

```
input heart_beat: Bool
output num_beats: Int8 @1Hz := heart_beat(over: 5s, count)
output num_beats_full: Int8 @1Hz := if time < 5 then dft else num_beats
trigger num_beats_full < 100
```

Here, `time` is an implicit stream containing the execution time of the monitor.

More formally, if an output stream $s_i^{\uparrow}$ with name `a` computes a sliding window with the new semantics, default value `dft` and window duration δ, another output stream $s_{i'}^{\uparrow}$, with the fresh name `a_full` is created. The new output stream mimics the pacing type of $s_i^{\uparrow}$ and has the stream expression `if time < δ then dft else a`. Any stream access to $s_i^{\uparrow}$ is replaced by an access to $s_{i'}^{\uparrow}$.

6.1.1. Type Omission

Type annotations indicate the shape of a data point. For input streams this is necessary to correctly identify how to interpret an input event. If the monitor receives a 32 bit wide event without information on what kind of value to expect, the same bit string can either represent 17.4 or 1099641651. For outputs, however, most types can be determined based on their expression. Consider the following specification:

```
input a: Float32
output b := a
```

The value type of `b` needs to be at least as restrictive as a $Float(32)$ because it accesses `a` synchronously, so both $Float(32)$ and $Float(64)$ are valid candidates. In contrast to that, the pacing type is uniquely defined as $\{s_1^{\downarrow}\}$. Thus, the specification contains a type

choice, a concept already discussed in Remark 4.3.2. By choosing the least restrictive choice, an explicit type annotation for `b` is not required to complete the type check. → Sec. 4.3, Page 36

It turns out, most of the types in an RTLola specification can be omitted without hindering the type analysis. This is because many expressions contain elements that narrow down the result type without requiring explicit type annotation. Consider, for example, a sliding window expression with a counting aggregation. The type of the aggregation is polymorphic:

$$count < T >\colon T^* \to \mathbb{N}$$

The resulting value thus cannot be a floating point number or a boolean value. Similarly, a conditional function is of type:

$$ite < T >\colon \mathbb{B} \times T \times T \to T$$

The first argument thus needs to resolve to a boolean value, and the second and third argument need to have the same type. Consider the following specification:

```
output s := if a.hold().defaults(to: false) then b else c.aggregate(over 3s, using: count)
```

Even without any further information on the streams `a`, `b`, or `c` we can deduce a lot of information:

- `a` is of value type *Bool* to be adequate for the conditional expression.
- `b` is of value type $UInt(X)$ for some valid X because its type needs to be compatible with the result of the aggregation.
- `c` also has an unsigned integer type that is at least as wide as `b` because of the type restrictions of the conditional.
- `s` is a periodic type because its expression contains a sliding window.
- `b` is periodic as well, because `s` accesses it synchronously. Moreover, its frequency needs to be a multiple of `s`'s frequency.

Type inference is an old and extensively studied field with relevance for both elderly languages like ML and modern ones like Rust and Swift. On of the most commonly used inference algorithms was presented by Roger Hindley in 1969 [67]. The interested reader can find more details there.

6.1.2. Full Syntax

The BNF for the full syntax including all of the syntactic sugar constructions can be found in the Appendix, Section A.2. → Sec. A.2, Page 86

6.2. Case Studies

Prior work on Lola and its extensions already applied the approach on network traffic [68, 4, 55] and unmanned aerial vehicles [69, 55]. In this theses, we present case studies from two different areas based on simulated or synthetic data.

In the first one we revise the case study for unmanned aerial vehicles presented by Adolf et al. [69]. They used Lola and thus lacked real-time components and aggregations. The former can be used to detect inconsistent timing in modules of the aircraft. These inconsistencies are indicators for deteriorating system health and can trigger compensation methods such as switching over to a spare module. Aggregations on the other hand allow for specifying complex properties such as a cross validation of different modules.

The second case study is concerned with monitoring medical cyber-physical systems, exemplified by implantable cardioverter-defibrillators (ICD) and responsive neurostimulators (RNS). ICDs are implants that monitor a patient's cardiac rhythm. Superficially speaking, it detects arrhythmias in which the heart does not follow its regular pattern. As a result, it cannot properly in- nor deflate and transitions into a state of ventricular fibrillation, resulting in a failure of pumping blood through the body. In such a case, the ICD triggers an electrical impulse that essentially stops the heart. This is supposed to stabilize the heart rhythm.[3] While arrhythmia detection is possible in theory, it proves hard in practice. This is due to strong variations in the heart cycle among different patients and natural variations e.g. induced by physical exertion. For this reason, specifying arrhythmias in languages without adequate primitives is extremely tedious. To make matters worse, misclassification by the ICD has dire consequences: False negatives, i.e., not detecting arrhythmias, can result in lethal cardiac arrest. False positives, i.e., administering shocks to a heart in a normal rhythm, leads to considerable pain in the best case, and fibrillation with subsequent cardiac arrest in the worst case. This indicates the importance of additional safety measures in the shape of a runtime monitor. We will investigate to which extend the respective properties can be expressed conveniently in RTLola.

The second kind of medical devices we discuss are responsive neurostimulators (RNS). These are implantable devices for patients suffering from drug resistant epilepsy. The devices monitors the electrophysiology (EEG) of the patient's brain via two leads that are inserted into the brain tissue [70]. The procedure is known as a cortical electroencephalogram. When the onset of an epileptic seizure is recognized based on abnormal brain activity, the implant administers mild electrical impulses. In many patients, this can prevent seizures entirely [71]. As opposed to the ICD, false positives, i.e. administering impulses due to misclassification, is not harmful.[4] The critical point behind this example is that the time between classification and response is only ap-

[3] This is contrary to popular believe that the shock "restarts the heart". A cardiac arrest resulting from a loss of heart function, i.e., the absence of impulses triggering the heart contraction, cannot be fixed by "shocking" the patient with a defibrillator.

[4] This holds true for short-term misclassification. The effects of long lasting administration of impulses is — to the best of our knowledge — not sufficiently studied.

proximately 3ms. Thus, there are strong timing constraints on the monitor for the RNS implant.

A common theme between all of these case studies is that there are hard requirements on the monitor — albeit to a different extend. The deployment of a monitor requires that it has low cost and resource consumption. Moreover, the correct behavior is safety critical: misjudgment leads to crashes, cardiac arrest or seizures.

In the following, we will present and discuss specifications for these case studies, as well as experimental results for running time and memory consumption. For this, we use an implementation of the language presented here, called StreamLab[5] [72].

Remark 6.2.1 (On Medical and Avionic Accuracy)**.** *The topics covered in this section are strongly simplified, glossing over a lot of details. Especially in the realm of medicine, we do not claim that the descriptions of the diseases are accurate. Rather, they are simplifications that allows us to break down the complex topic to a level that makes it presentable in this thesis. Accurate specifications require a consideration of a myriad of corner cases that vary from patient to patient and from aircraft to aircraft. Moreover, we want to shift the focus to the language rather than details about epilepsy or flight control.*

6.2.1. Helper Macros

For the specifications in the upcoming sections, we use macros, i.e. domain specific syntactic sugar as follows:

```
δ(s) ≡ (s - s.offset(by: -1).defaults(to: s))
∇(s) ≡ δ(s) / (if δ(time) = 0 then 1 else δ(time)
∇ⁿ(s) ≡ ∇(∇ⁿ⁻¹(s))
```

The first expression, `δ(s)`, computes the difference between the current and the last reading of stream `s` and defaults to 0 if `s` only has a single value. The second expression, `∇(s)`, differentiates `s` discretely. The conditional sub-expression guarantees that the denominator is non-zero in the first evaluation phase when `δ(time)` is 0. The last macro applies the discrete derivation n times for some specification-time constant $n \in \mathbb{N}$.

6.2.2. Unmanned Aerial Vehicles

The input data for this case study was generated with the state-of-the-art software-in-the-loop framework ArduPilot[6] using the drone autopilot ArduCopter[7]. The simulations generated a trace of 432.961 input events containing the longitude (`lon`), latitude (`lat`), wind direction (`wnd_dir`), and wind speed (`wnd_spd`). The trace was annotated with valid and spurious command signals afterwards.

The full specification for the unmanned aerial vehicles can be found in Figure 6.1. It consists of the declarations for input streams (lines 1–4) in correspondence to the generated log data.

[5] `http://stream-lab.eu`
[6] `http[://ardupilot.org/`
[7] `http://ardupilot.org/copter/`

Lines 6–7 check that the GPS module provides data in a frequency of at least 5Hz by counting the number of events from the `lat` stream for one second. If the value drops below 5, the trigger raises an alarm. Note that the sliding window for the lower bound is a conservative one: the first second of the mission, the window expression does not yield a result. This avoids spurious warnings at the beginning of the execution when the GPS did not have a chance to generate five data points, yet.

Lines 9–11 cross validate the GPS module against the velocity generated by the inertia measurement unit (IMU). For this, `gps_dist` computes the traveled distance using the Pythagorean theorem. A more realistic specification would use the haversine formula to account for the curvature of the earth. This example, however, uses the simplified version disregarding the resulting inaccuracy. The stream `gps_velo` then computes the velocity based on the GPS readings by differentiating the distance discretely. Lastly, the trigger compares the GPS velocity against the IMU velocity. If the deviation exceeds a threshold, an alarm is raised.

Lines 13-16 and Lines 18-24 check for spurious slow-down and hover maneuvers, respectively. For this, the specification checks whether a maneuver was executed, and if so, it checks whether the respective command was given within the last 5s. For this, the specification computes a sliding window over the boolean command stream with an existential quantification as aggregation function. The detection of a slow-down is straight-forward: if the velocity was high before and dropped below a threshold, the aircraft is decelerating. The detection of a hover phase checks the spatial progress of the aircraft. Hereby, it distinguishes between a deliberate standstill and one that results from strong headwind. For the headwind detection, the current flight direction `dir` is computed as the inverse tangent of the latitude divided by the longitude. The wind is considered a headwind if the flight and wind direction are sufficiently similar and the wind speed sufficiently high. A hovering phase then manifests as a time in which there is no headwind and the covered distance, i.e., the integral of the velocity, is sufficiently low. Lastly, the trigger condition is analogous to the one for the slow-down detection.

Evaluating the entire specification with StreamLab takes 1.42s for the aforementioned 432.961 input events, which induced 865.920 relevant timestamps. Broken down, this means that the evaluation of a single relevant timestamp amounted to 1435ns. The monitor consumed 15.55MB memory; in addition to the memory configuration, this includes the internal representation of the specification, the program code and parts of the C standard library. The reason for this is that the monitor is implemented in rust, which is statically linked against the C standard library. The stack size remained below 1kB, which shows that a) the call depth is shallow and b) there is little allocation during the execution. In fact, after the initialization there is no dynamic allocation anymore resulting in a constant heap size.

In conclusion, this case study shows that even complex specifications such as "A time frame with little spatial progress despite non-malicious wind conditions must be preceded by a hovering command" can be expressed in an understandable way in RTLola. The resulting monitor evaluates events in the realm of microseconds with manageable memory overhead.

```
1  input lat, lon, velo, time: Float64
2  input slow_down_cmd: Bool
3  input hover_cmd: Bool
4  input wnd_dir, wnd_spd : Float32
5
6  output gps_freq @1Hz := lat.aggregate(over_exactly: 1s, using: count).defaults(to: 5)
7  trigger gps_freq < 5 "GPS frequency less than 5 Hz"
8
9  output gps_dist := sqrt(δ(lon)^2 + δ(lat)^2)
10 output gps_velo := ∇(gps_dist)
11 trigger abs(gps_velo - velo) > 0.1 "Conflicting measurements for velocity."
12
13 output fast := velo > 700
14 output slow_down := fast.offset(by:-1).defaults(to:false) ∧ ¬fast
15 trigger @1Hz ¬slow_down_cmd.aggregate(over: 5s, using: ∃)
16            ∧ slow_down.hold().defaults(to: false) "Spurious Slow-Down."
17
18 output dir := arctan(δ(lat)/δ(lon))
19 output headwind := abs(wnd_dir - dir) < 0.2 ∧ wnd_spd > 10.0
20 output hovering @1Hz := velo.aggregate(over: 5s, using: ∫) < 0.5
21                       ∧ ¬headwind.hold().defaults(to: false)
22 trigger @1Hz ¬hover_cmd.aggregate(over: 5s, using: ∃)
23            ∧ hovering.hold().defaults(to: false)
24            "Spurious Hovering. Path planner hung up?"
```

Figure 6.1.: Specification for the unmanned aerial vehicle case study. The triggers detect low sampling rates in the GPS module, deviations in location approximations, and spurious slow down and hovering phases.

6.2.3. Medical Cyber-Physical Systems

Unfortunately, there is no accessible simulation framework for medical data qualitatively comparable to ArduPilot. So this case study uses synthetic data, which is sufficient as a proof of concept.

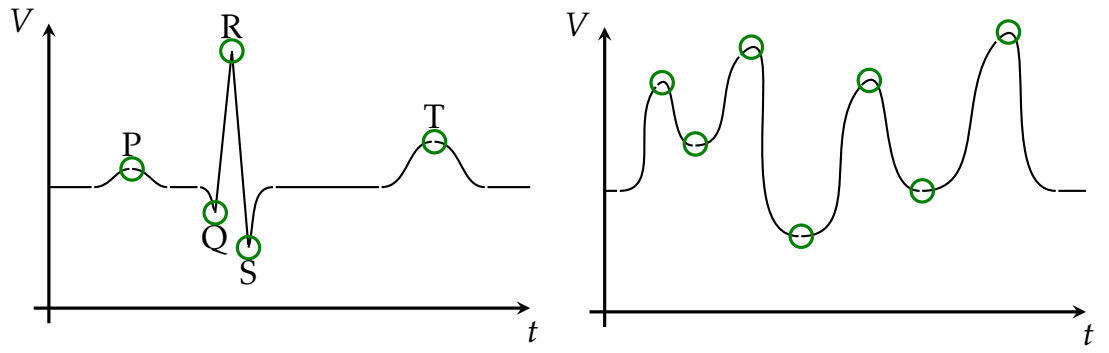
Cardiac Arrhythmia

We start by discussing the specification for detecting cardiac arrhythmia, in particular ventricular fibrillation (V-fib). For this, consider the electro cardiogram (EKG) of a regular sinus rhythm in Figure 6.2a. The rhythm consists of five clearly distinguishable phases. The key factors for the specification are the extrema, indicated in green, which can be recognized by a zero crossing in the derivation of the signal. Note that the detection of roots is insufficient due to the expected turning point between phase S and T. We present a simplified anomaly detection algorithm that yields an indicator of an arrhythmia. For a reliable analysis, several such criteria need to be monitored and the classification checks whether a sufficient amount of them are met. The presented method identifies the extrema that indicates phase switches and checks their potential in relation to the extrema of other phases. If, for example, the local extremum of the T phase exceeds the potential of the extremum of the R phase, one of the two phases were irregular. Upon inspection of the arrhythmic EKG in Figure 6.2b, one can see that such comparisons between peeks will fail because the wave does not follow the expected phase pattern.

The specification in Figure 6.3 computes these comparisons. The `cop` (change of potential) stream contains the first derivative of the potential. Lines 6–8 determine the point in time when a zero crossing takes place. The stream `state` contains the current state. Note that we write P–T for readability, they stand for the integer values 0–4, respectively. `local_extreme` is a stream that is 0.0 unless a zero crossing took place in the same evaluation cycle, in which case it contains the extreme value. Lines 19 and 20 use this information to compute the greatest and least value over the last 2s. This assumes that the patient has a heart rate of approximately 60. In a realistic scenario, the specification would have to take the heart rate into account and adapt its outputs accordingly.

The arrhythmia detection then checks whether a zero_crossing took place. If so, it further checks the current phase using the `state` stream and compares the extreme value against the computed time-bounded extrema. Lastly, the trigger counts the number of detected pattern violations and raises an alarm if seven out of the estimated ten heart beats exhibit arrhythmic behavior.

Monitoring the specification with StreamLab yields similar results to the first case study: On average, each one of the 808.000 relevant timestamps required 1420ns, the stack size never exceeded 1kB. The required memory, however, decreased to 11.09MB due to the lower amount of less memory-intense sliding windows: for comparison, an integral window requires approximately four times the amount of memory of an extremum-aggregation.



(a) The EKG of a regular sinus rhythm.

(b) The EKG of an irregular cardiac rhythm.

Figure 6.2.: EKGs for a regular sinus rhythm (left) and an arrhythmia resulting in V-fib (right). The local extrema of the signals and thus zero crossings of the derivative are marked in green. The five phases P through T of the regular heartbeat are clearly distinguishable in the sinus rhythm. In the arrhythmic case, the phases become indistinct.

Drug-Resistant Epilepsy

The last example presented in this thesis is concerned with the detection and prevention of epileptic seizures. Figure 6.4 shows the characteristic electroencephalogram (EEG) of a seizure patient. The measured potential underlies natural fluctuation. During the onset of a seizure, the amplitude of the fluctuation rises significantly. The implanted responsive neurostimulator administers electric impulses upon detection of the abnormal increase. The short flat line followed by a spike in the EEG indicate the stimulation. If the brain wave does not normalize afterwards, additional impulses can be administered until the seizure ceases. This effectively prevents a seizure: the pattern shown in Figure 6.4 typically spans less than 20ms. For comparison, the average reaction time to auditory stimuli is around 150ms [73].

The specification for monitoring the implant can be found in Figure 6.5. While the implant has two leads, we simplify the specification by only considering one, the Cortical Strip Lead (CSL). In addition, the monitor gets two boolean inputs from the implant, one indication that it `rec`(ognized) an onset, and one indicating the administration of a `stim`(ulation). The specification first computes the absolute *jerk*, i.e. the third derivative, of the measured potential. Intuitively speaking, a high jerk represents an unsteady signal. In addition to that, we compute the long-term average jerk over 2000s as reference value and the short-term average jerk over 2ms. If the short-term average exceeds the long-term average by a margin of ϵ, an unsteady signal is detected. Lines 10–13 now monitor the response of the implant: it checks whether an onset recognized by the monitor was not detected by the implant and whether a recognized onset was not followed by stimulation.

```
input potential: Float64
input time: Float64

output cop := ∇(potential)

output pos_to_neg := (sign(cop.offset(by: -1).defaults(to: 0.0) = 1) ∧ sign(cop) = -1
output neg_to_pos := (sign(cop.offset(by: -1).defaults(to: 0.0) = -1 ∧ sign(cop) = 1
output zero_crossing := neg_to_pos ∨ pos_to_neg

output state :=
  if zero_crossing
  then (state.offset(by: -1).defaults(to: P) + 1) % 5
  else state.offset(by: -1).defaults(to: P)

output local_extreme :=
  if zero_crossing
  then potential.offset(by: -1).defaults(to: 1.0)
  else 0.0
output bounded_max @1Hz :=
  local_extreme.aggregate(over: 2s, using: max).defaults(to: 0.0)
output bounded_min @1Hz :=
  local_extreme.aggregate(over: 2s, using: min).defaults(to: 0.0)

output arrhythmic := zero_crossing ∧
    (
     (state = P ∨ state = T) ∧ local_extreme > bounded_max.hold().defaults(to: 0)
     |
     state = R ∧ local_extreme < bounded_max.hold().defaults(to: 0.0) * 0.7
     |
     (state = Q ∨ state = S) ∧ local_extreme > bounded_min.hold().defaults(to: 0.0)
    )

trigger @10Hz arrhythmic.aggregate(over_exactly: 10s, using: sum).defaults(to: 0) > 7
    "Irregular heart beat detected."
```

Figure 6.3.: A specification detecting cardiac arrhythmia via comparing extrema of the measured potential.

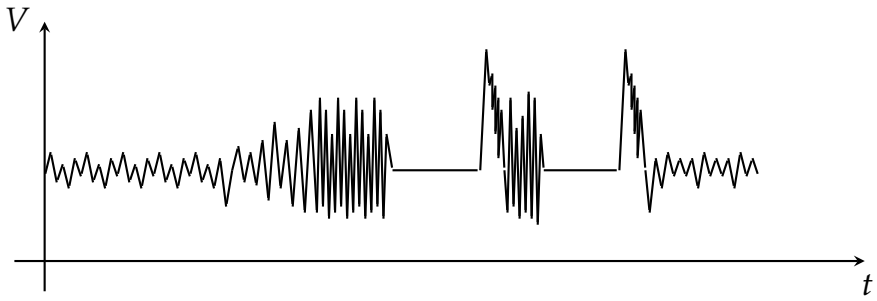


Figure 6.4.: An EEG showing the onset of an epileptic seizure. The short flat line followed by a spike indicates a neural stimulation as countermeasure against the seizure.

```
1  input CSL: Float64
2  input rec, stim: Bool
3
4  output jerk := abs(∇³(CLS))
5
6  output avg_long @100mHz := jerk.aggr(over: 2000s, using: avg)
7  output avg_short @1kHz := jerk.aggr(over: 2ms, using: avg)
8  output spike @1kHz := avg_short > avg_long.hold() + ϵ
9
10 trigger @1kHz spike ∧ ¬rec.aggr(over: 2ms, using: ∃)
11                          "seizure not recognized"
12 trigger @1kHz rec.aggr(over: 5ms, using: ∃) ∧ ¬stim.aggr(over: 3ms, using: ∃)
13                          "stimulation not triggered"
```

Figure 6.5.: Specification monitoring a responsive neurostimulator based on the potential measured with the Cortical Strip Lead.

Evaluating a synthetic event sequence with StreamLab showed that — despite the stringent time constraints — the monitor was able to keep up: evaluating a single event took around 3µs. This is considerably higher than for the other case studies despite the specification's shorter length. The reason behind this is the ratio of duration and evaluation frequency in the sliding window expressions. They lead to a significantly greater amount of intermediate values that need to be managed and aggregated. This also impacts the memory consumption: the intermediate values and bookkeeping data structures add a little less than 3MB to the total required memory of 14.02MB. If this overhead exceeds the available resources of the implant, reducing the durations of the windows or evaluation frequency also reduces the required memory. Especially the long-term average allows for some adjustments without loss of correctness.

Chapter 7

Final Remarks and Future Directions

This thesis presented formal semantics for the specification language RTLola coupled with an evaluation algorithm. The main advantages of the language is the expressiveness that allows to specify complex properties in an understandable way, and static guarantees on the running time and memory overhead. With these advantages we propose to use monitors for RTLola specifications in combination with complex, unverified control structures. This especially includes machine learned controllers. They grow ever more popular but their reasoning remains too convoluted to be understood by humans. However, detailed understanding is not necessary when the confidence in the runtime behavior is sufficiently high while a trusted component monitors the behavior. In case of an emergency, a verified, inefficient safety controller can take over and stabilize the system execution.

This might pave the way towards certifiable autonomous controllers for vehicles as well as new kinds of medical implants. In both of these fields, the certifiability plays a crucial role and it is yet unclear how to achieve it with learned components. Despite that, their outstanding performance is too convincing to pass up on. So runtime monitoring with RTLola can be the connecting link between efficiency and certifiability.

However, for this, the subject of traceability needs to be discussed for RTLola. A monitor is traceable if their decision making is evident without detailed knowledge about the implementation. The current implementation of StreamLab, however, is an interpreter for RTLola specifications. Interpretation is abstract, which hinders traceability. A compilation to FPGA [55] is the first step in this direction, but a thorough investigation of compilation-based approaches is still necessary.

In addition to that, the language is not yet complete. State machines play a huge role in the development of complex systems. Without special syntax, however, their encoding in RTLola leaves a lot to be desired. Similarly, domain specific constructs can simplify specifications significantly. For this, a library-based approach offers itself:

through including a library e.g. for network monitoring or avionics, specific functions and macros become accessible. Moreover, activation conditions are an interesting concept for future iterations of RTLola. An activation condition is a condition on the current state of the monitor that indicates whether a stream should be extended or not. While this thesis discussed the basic, type-based form of activation conditions, boolean conditions evaluated during runtime are also an option. This way, a stream can forgo being evaluated if certain criteria are not met. Therefore, the stream essentially filters itself.

In conclusion, we consider RTLola a valuable tool for the development of safety-critical systems and we will strive to extend the framework on the theoretical side as well as deploy it in real-world applications.

Appendix A

Appendix

A.1. BNF for the Concrete Syntax

⟨*spec*⟩ ::= ⟨*input-list*⟩ ⟨*output-list*⟩ ⟨*trigger-list*⟩

⟨*input-list*⟩ ::= ⟨*input*⟩ ⟨*input-list*⟩ | ϵ

⟨*output-list*⟩ ::= ⟨*output*⟩ ⟨*output-list*⟩ | ⟨*output*⟩

⟨*trigger-list*⟩ ::= ⟨*trigger*⟩ ⟨*trigger-list*⟩ | ϵ

⟨*input*⟩ ::= ‘`input`’ ⟨*name*⟩ ‘`:`’ ⟨*vtype*⟩

⟨*output*⟩ ::= ‘`output`’ ⟨*name*⟩ ‘`:`’ ⟨*vtype*⟩ := ⟨*expr*⟩
| ‘`output`’ ⟨*name*⟩ ‘`:`’ ⟨*vtype*⟩ ‘`@`’ ⟨*ptype*⟩ := ⟨*expr*⟩

⟨*trigger*⟩ ::= ‘`trigger`’ ⟨*name*⟩ ‘`"`’⟨*msg*⟩‘`"`’

⟨*vtype*⟩ ::= ‘`Bool`’ | ‘`Int`’⟨*int-width*⟩ | ‘`UInt`’⟨*int-width*⟩ | ‘`Float`’⟨*fl-width*⟩

⟨*int-width*⟩ ::= 8 | 16 | 32 | 64

⟨*fl-width*⟩ ::= 32 | 64

⟨*ptype*⟩ ::= ⟨*number*⟩ ‘`Hz`’

⟨*expr*⟩ ::= ⟨*name*⟩(⟨*expr-list*⟩)
| ⟨*name*⟩‘`.offset(by:`’ ⟨*integer*⟩ ‘`)`’
| ⟨*name*⟩‘`.defaults(to:`’ ⟨*expr*⟩ ‘`)`’
| ⟨*name*⟩
| ⟨*name*⟩‘`.hold()`’
| ⟨*name*⟩‘`.aggregate(over:`’ ⟨*integer*⟩‘`s, using:`’ ⟨*name*⟩‘`)`’

⟨*expr-list*⟩ ::= ⟨*expr*⟩ ⟨*expr-list*⟩ | ϵ

⟨*name*⟩ ::= ⟨*letter*⟩⟨*alphanum*⟩*

⟨*letter*⟩ ::= a–z | A–Z

⟨*digit*⟩ ::= 0–9

⟨*integer*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩⟨*integer*⟩

⟨*alphanum*⟩ ::= ⟨*letter*⟩ | ⟨*digit*⟩

A.2. BNF for the Sugarized Syntax

⟨*spec*⟩ ::= ⟨*declaration list*⟩ ⟨*output*⟩ ⟨*declaration list*⟩

⟨*declaration-list*⟩ ::= ⟨*declaration*⟩ ⟨*declaration-list*⟩ | ϵ

⟨*declaration*⟩ ::= ⟨*input*⟩ | ⟨*output*⟩ | ⟨*trigger*⟩

⟨*input*⟩ ::= '`input`' ⟨*name*⟩ '`:`' ⟨*vtype*⟩

⟨*output*⟩ ::= '`output`' ⟨*name*⟩ '`:`' ⟨*vtype*⟩ '`@`' ⟨*ptype*⟩ := ⟨*expr*⟩
| '`output`' ⟨*name*⟩ '`:`' ⟨*vtype*⟩ := ⟨*expr*⟩
| '`output`' ⟨*name*⟩ '`@`' ⟨*ptype*⟩ := ⟨*expr*⟩
| '`output`' ⟨*name*⟩ := ⟨*expr*⟩

⟨*trigger*⟩ ::= '`trigger @`' ⟨*ptype*⟩ ⟨*expr*⟩ ⟨*trig-msg*⟩
| '`trigger`' ⟨*expr*⟩ ⟨*trig-msg*⟩

⟨*trig-msg*⟩ ::= '`"`'⟨*msg*⟩'`"`' | ϵ

⟨*vtype-list*⟩ ::= ⟨*vtype*⟩ ⟨*vtype-list*⟩ | ϵ

⟨*vtype*⟩ ::= ⟨*avtype*⟩ | (⟨*vtype-list*⟩)

⟨*avtype*⟩ ::= '`Bool`' | '`Int`'⟨*int-width*⟩ | '`UInt`'⟨*int-width*⟩ | '`Float`'⟨*fl-width*⟩

⟨*int-width*⟩ ::= 8 | 16 | 32 | 64

⟨*fl-width*⟩ ::= 32 | 64

⟨*ptype*⟩ ::= ⟨*number*⟩ '`Hz`'

⟨*expr*⟩ ::= ⟨*name*⟩.⟨*name*⟩(⟨*expr-list*⟩)
| ⟨*name*⟩‘`.offset(by:`’ ⟨*integer*⟩ ‘`)`’
| ⟨*name*⟩‘`.defaults(to:`’ ⟨*expr*⟩ ‘`)`’
| ⟨*name*⟩
| ⟨*name*⟩‘`.hold()`’
| ⟨*name*⟩‘`.aggregate(over:`’ ⟨*integer*⟩‘`s, using:`’ ⟨*name*⟩‘`)`’
| ‘`if`’ ⟨*expr*⟩ ‘`then`’ ⟨*expr*⟩ ‘`else`’ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*binop*⟩ ⟨*expr*⟩
| ⟨*unop*⟩ ⟨*expr*⟩

⟨*binop*⟩ ::= ‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`/`’ | ‘`%`’ | ‘`&`’ | ‘`|`’ | ‘`=`’ | ‘`>`’ | ... ⟨*unop*⟩ ::= ‘`-`’ | ‘`!`’ | ...

⟨*expr-list*⟩ ::= ⟨*expr*⟩ ⟨*expr-list*⟩ | ϵ

⟨*name*⟩ ::= ⟨*letter*⟩⟨*alphanum*⟩*

⟨*letter*⟩ ::= a–z | A–Z

⟨*digit*⟩ ::= 0–9

⟨*integer*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩⟨*integer*⟩

⟨*alphanum*⟩ ::= ⟨*letter*⟩ | ⟨*digit*⟩

Here, <msg> can be any sequence of symbols excluding double quotation marks (");

Bibliography

[1] J. S. Bell. 1964. On the Einstein-Podolsky-Rosen paradox. *Physics Physique Fizika*, 1, 195–200. DOI: 10.1103/PhysicsPhysiqueFizika.1.195.

[2] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, 46–57. DOI: 10.1109/SFCS.1977.32. https://doi.org/10.1109/SFCS.1977.32.

[3] Andreas Bauer, Martin Leucker, and Christian Schallhart. 2011. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20, 4, 14:1–14:64. DOI: 10.1145/2000799.2000800.

[4] Peter Faymonville. 2019. *Monitoring with Parameters*. PhD thesis. Saarland University, Saarbrücken, Germany. https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/27458.

[5] Howard Barringer, David E. Rydeheard, and Klaus Havelund. 2010. Rule systems for run-time monitoring: from Eagle to RuleR. *J. Log. Comput.*, 20, 3, 675–706. DOI: 10.1093/logcom/exn076.

[6] Martin Roesch. 1999. Snort: lightweight intrusion detection for networks. In *LISA 1999*. David W. Parter, editor. USENIX, 229–238. ISBN: 1-880446-25-1.

[7] Doron Drusinsky. 2000. The temporal rover and the ATG rover. In *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings* (Lecture Notes in Computer Science). Klaus Havelund, John Penix, and Willem Visser, editors. Volume 1885. Springer, 323–330. ISBN: 3-540-41030-9. DOI: 10.1007/10722468_19. https://doi.org/10.1007/10722468_19.

[8] Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. 1999. Runtime assurance based on formal specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 1999, June 28 - Junlly 1, 1999, Las Vegas, Nevada, USA*. Hamid R. Arabnia, editor. CSREA Press, 279–287. ISBN: 1-892512-15-7.

[9] Bernd Finkbeiner and Henny Sipma. 2004. Checking finite traces using alternating automata. *Formal Methods in System Design*, 24, 2, 101–127. DOI: 10.1023/B:FORM.0000017718.28096.48. https://doi.org/10.1023/B:FORM.0000017718.28096.48.

[10] Orna Kupferman and Moshe Y. Vardi. 2001. Model checking of safety properties. *Formal Methods in System Design*, 19, 3, 291–314. DOI: 10.1023/A:1011254632723. https://doi.org/10.1023/A:1011254632723.

[11] Klaus Havelund and Grigore Rosu. 2002. Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings* (LNCS). Joost-Pieter Katoen and Perdita Stevens, editors. Volume 2280. Springer, 342–356. ISBN: 3-540-43419-4. DOI: 10.1007/3-540-46002-0_24. https://doi.org/10.1007/3-540-46002-0_24.

[12] Ron Koymans. 1990. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2, 4, 255–299. DOI: 10.1007/BF01995674. https://doi.org/10.1007/BF01995674.

[13] Oded Maler and Dejan Nickovic. 2004. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings* (LNCS). Yassine Lakhnech and Sergio Yovine, editors. Volume 3253. Springer, 152–166. ISBN: 3-540-23167-6. DOI: 10.1007/978-3-540-30206-3_12. https://doi.org/10.1007/978-3-540-30206-3_12.

[14] Jyotirmoy V. Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A. Seshia. 2017. Robust online monitoring of signal temporal logic. *Formal Methods in System Design*, 51, 1, 5–30. DOI: 10.1007/s10703-017-0286-7. https://doi.org/10.1007/s10703-017-0286-7.

[15] Dejan Nickovic and Oded Maler. 2007. AMT: A property-based monitoring tool for analog systems. In *Formal Modeling and Analysis of Timed Systems, 5th International Conference, FORMATS 2007, Salzburg, Austria, October 3-5, 2007, Proceedings*, 304–319. DOI: 10.1007/978-3-540-75454-1_22. https://doi.org/10.1007/978-3-540-75454-1_22.

[16] David A. Basin, Felix Klaedtke, Samuel Müller, and Eugen Zalinescu. 2015. Monitoring metric first-order temporal properties. *J. ACM*, 62, 2, 15:1–15:45. DOI: 10.1145/2699444. https://doi.org/10.1145/2699444.

[17] David A. Basin, Srdjan Krstic, and Dmitriy Traytel. 2017. AERIAL: almost event-rate independent algorithms for monitoring metric regular properties. In *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017,*

Seattle, WA, USA (Kalpa Publications in Computing). Giles Reger and Klaus Havelund, editors. Volume 3. EasyChair, 29–36. http://www.easychair.org/publications/paper/sgWQ.

[18] Marc Boule and Zeljko Zilic. 2008. Automata-based assertion-checker synthesis of PSL properties. *ACM Trans. Design Autom. Electr. Syst.*, 13, 1, 4:1–4:21. DOI: 10.1145/1297666.1297670. https://doi.org/10.1145/1297666.1297670.

[19] Anat Dahan, Daniel Geist, Leonid Gluhovsky, Dmitry Pidan, Gil Shapir, Yaron Wolfsthal, Lyes Benalycherif, Romain Kamdem, and Younes Lahbib. 2005. Combining system level modeling with assertion based verification. In *6th International Symposium on Quality of Electronic Design (ISQED 2005), 21-23 March 2005, San Jose, CA, USA*, 310–315. DOI: 10.1109/ISQED.2005.32. https://doi.org/10.1109/ISQED.2005.32.

[20] Ping Hang Cheung and Alessandro Forin. 2007. A c-language binding for PSL. In *Embedded Software and Systems, [Third] International Conference, ICESS 2007, Daegu, Korea, May 14-16, 2007, Proceedings*, 584–591. DOI: 10.1007/978-3-540-72685-2_54. https://doi.org/10.1007/978-3-540-72685-2_54.

[21] Rodolfo Pellizzoni, Patrick O'Neil Meredith, Marco Caccamo, and Grigore Rosu. 2008. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain, 30 November - 3 December 2008*, 481–491. DOI: 10.1109/RTSS.2008.43. https://doi.org/10.1109/RTSS.2008.43.

[22] Stefan Jaksic, Ezio Bartocci, Radu Grosu, Reinhard Kloibhofer, Thang Nguyen, and Dejan Nickovic. 2015. From signal temporal logic to FPGA monitors. In *13. ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, Austin, TX, USA, September 21-23, 2015*, 218–227. DOI: 10.1109/MEMCOD.2015.7340489. https://doi.org/10.1109/MEMCOD.2015.7340489.

[23] Patrick Moosbrugger, Kristin Y. Rozier, and Johann Schumann. 2017. R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. *Formal Methods in System Design*, 51, 1, 31–61. DOI: 10.1007/s10703-017-0275-x. https://doi.org/10.1007/s10703-017-0275-x.

[24] Patrick Moosbrugger, Kristin Y. Rozier, and Johann Schumann. 2017. R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. *Formal Methods in System Design*, 51, 1, 31–61. DOI: 10.1007/s10703-017-0275-x. https://doi.org/10.1007/s10703-017-0275-x.

[25] Klaus Havelund, Doron Peled, and Dogan Ulus. 2017. First order temporal logic monitoring with BDDs. In *FMCAD 2017*. Daryl Stewart and Georg Weissenbacher, editors. IEEE, 116–123. ISBN: 978-0-9835678-7-5. DOI: 10.23919/FMCAD.2017.8102249.

[26] David A. Basin, Matús Harvan, Felix Klaedtke, and Eugen Zalinescu. 2011. MONPOLY: monitoring usage-control policies. In *RV 2011* (LNCS). Sarfraz Khurshid and Koushik Sen, editors. Volume 7186. Springer, 360–364. ISBN: 978-3-642-29859-2. DOI: 10.1007/978-3-642-29860-8_27.

[27] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. 2004. Rule-based runtime verification. In *VMCAI 2004* (LNCS). Bernhard Steffen and Giorgio Levi, editors. Volume 2937. Springer, 44–57. ISBN: 3-540-20803-8. DOI: 10.1007/978-3-540-24622-0_5.

[28] Klaus Havelund. 2015. Rule-based runtime verification revisited. *STTT*, 17, 2, 143–170. DOI: 10.1007/s10009-014-0309-2.

[29] Normann Decker, Martin Leucker, and Daniel Thoma. 2016. Monitoring modulo theories. *STTT*, 18, 2, 205–225. DOI: 10.1007/s10009-015-0380-3.

[30] Stefan Jaksic, Ezio Bartocci, Radu Grosu, Thang Nguyen, and Dejan Nickovic. 2018. Quantitative monitoring of STL with edit distance. *Formal Methods in System Design*, 53, 1, 83–112. DOI: 10.1007/s10703-018-0319-x.

[31] Bernd Finkbeiner and Hazem Torfah. 2017. The density of linear-time properties. In *ATVA 2017* (LNCS). Deepak D'Souza and K. Narayan Kumar, editors. Volume 10482. Springer, 139–155. ISBN: 978-3-319-68166-5. DOI: 10.1007/978-3-319-68167-2_10.

[32] Hazem Torfah and Martin Zimmermann. 2018. The complexity of counting models of linear-time temporal logic. *Acta Inf.*, 55, 3, 191–212. DOI: 10.1007/s00236-016-0284-z.

[33] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. 2012. Quantified event automata: towards expressive and efficient runtime monitors. In *FM 2012* (LNCS). Dimitra Giannakopoulou and Dominique Méry, editors. Volume 7436. Springer, 68–84. ISBN: 978-3-642-32758-2. DOI: 10.1007/978-3-642-32759-9_9.

[34] Ramy Medhat, Borzoo Bonakdarpour, Sebastian Fischmeister, and Yogi Joshi. 2016. Accelerated runtime verification of LTL specifications with counting semantics. In *RV 2016* (LNCS). Yliès Falcone and César Sánchez, editors. Volume 10012. Springer, 251–267. ISBN: 978-3-319-46981-2. DOI: 10.1007/978-3-319-46982-9_16.

[35] Simone Silvetti, Laura Nenzi, Ezio Bartocci, and Luca Bortolussi. 2018. Signal convolution logic. In *ATVA 2018*, 267–283. DOI: 10.1007/978-3-030-01090-4_16.

[36] Corto Mascle, Daniel Neider, Maximilian Schwenger, Paulo Tabuada, Alexander Weinert, and Martin Zimmermann. 2018. From ltl to rltl monitoring: improved monitorability through robust semantics. *CoRR*, abs/1807.08203. arXiv: 1807.08203. http://arxiv.org/abs/1807.08203.

[37] Paulo Tabuada and Daniel Neider. 2016. Robust linear temporal logic. In *CSL 2016* (LIPIcs). Jean-Marc Talbot and Laurent Regnier, editors. Volume 62. Schloss Dagstuhl - LZI, 10:1–10:21. ISBN: 978-3-95977-022-4. DOI: 10.4230/LIPIcs.CSL.2016.10.

[38] Tzanis Anevlavis, Daniel Neider, Matthew Phillipe, and Paulo Tabuada. 2019. Evrostos: the rLTL verifier. In *HSCC 2019*. Necmiye Ozay and Pavithra Prabhakar, editors. ACM, 218–223. ISBN: 978-1-4503-6282-5. DOI: 10.1145/3302504.3311812.

[39] David A. Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zalinescu. 2015. Monitoring of temporal first-order properties with aggregations. *Formal Methods in System Design*, 46, 3, 262–285. DOI: 10.1007/s10703-015-0222-7.

[40] Takumi Akazaki and Ichiro Hasuo. 2015. Time robustness in MTL and expressivity in hybrid system falsification. In *CAV 2015* (LNCS). Daniel Kroening and Corina S. Pasareanu, editors. Volume 9207. Springer, 356–374. ISBN: 978-3-319-21667-6. DOI: 10.1007/978-3-319-21668-3_21.

[41] Houssam Abbas, Alena Rodionova, Ezio Bartocci, Scott A. Smolka, and Radu Grosu. 2017. Quantitative regular expressions for arrhythmia detection algorithms. In *CMSB 2017* (LNCS). Jérôme Feret and Heinz Koeppl, editors. Volume 10545. Springer, 23–39. ISBN: 978-3-319-67470-4. DOI: 10.1007/978-3-319-67471-1_2.

[42] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. 2016. Regular programming for quantitative properties of data streams. In *ESOP 2016* (LNCS). Peter Thiemann, editor. Volume 9632. Springer, 15–40. ISBN: 978-3-662-49497-4. DOI: 10.1007/978-3-662-49498-1_2.

[43] Bernd Finkbeiner, Sriram Sankaranarayanan, and Henny Sipma. 2005. Collecting statistics over runtime executions. *Form. Meth. in Sys. Des.*, 27, 3, 253–274. DOI: 10.1007/s10703-005-3399-3.

[44] Sylvain Hallé. 2016. When RV meets CEP. In *RV 2016* (LNCS). Yliès Falcone and César Sánchez, editors. Volume 10012. Springer, 68–91. ISBN: 978-3-319-46981-2. DOI: 10.1007/978-3-319-46982-9_6.

[45] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. 2010. Copilot: A hard real-time runtime monitor. In *RV 2010* (LNCS). Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors. Volume 6418. Springer, 345–359. ISBN: 978-3-642-16611-2. DOI: 10.1007/978-3-642-16612-9_26.

[46] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. 1987. Lustre: A declarative language for programming synchronous systems. In *POPL 1987*. ACM Press, 178–188. ISBN: 0-89791-215-2. DOI: 10.1145/41625.41641.

[47] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. 2005. LOLA: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA*. IEEE Computer Society, 166–174. ISBN: 0-7695-2370-6. DOI: 10.1109/TIME.2005.26. https://doi.org/10.1109/TIME.2005.26.

[48] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. 1987. 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, 178–188. ISBN: 0-89791-215-2. DOI: 10.1145/41625.41641. https://doi.org/10.1145/41625.41641.

[49] Nicolas Halbwachs. 2005. A synchronous language at work: the story of lustre. In *3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2005), 11-14 July 2005, Verona, Italy, Proceedings*, 3–11. DOI: 10.1109/MEMCOD.2005.1487884. https://doi.org/10.1109/MEMCOD.2005.1487884.

[50] Gérard Berry and Georges Gonthier. 1992. The esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19, 2, 87–152. DOI: 10.1016/0167-6423(92)90005-V. https://doi.org/10.1016/0167-6423(92)90005-V.

[51] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. 2005. LOLA: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA*. IEEE Computer Society, 166–174. ISBN: 0-7695-2370-6. DOI: 10.1109/TIME.2005.26. https://doi.org/10.1109/TIME.2005.26.

[52] Lukas Convent, Sebastian Hungerecker, Torben Scheffel, Malte Schmitz, Daniel Thoma, and Alexander Weiss. 2018. Hardware-based runtime verification with embedded tracing units and stream processing. In *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings* (Lecture Notes in Computer Science). Christian Colombo and Martin Leucker, editors. Volume 11237. Springer, 43–63. ISBN: 978-3-030-03768-0. DOI: 10.1007/978-3-030-03769-7_5. https://doi.org/10.1007/978-3-030-03769-7_5.

[53] Felipe Gorostiaga and César Sánchez. 2018. Striver: stream runtime verification for real-time event-streams. In *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings* (Lecture Notes in Computer Science). Christian Colombo and Martin Leucker, editors. Volume 11237. Springer, 282–298. ISBN: 978-3-030-03768-0. DOI: 10.1007/978-3-030-03769-7_16. https://doi.org/10.1007/978-3-030-03769-7_16.

[54] Marcel Maltry. 2017. *FPGA-based Monitoring for Stream Specification Languages*. Master's thesis. Saarland University, (July 2017).

[55] Jan Baumeister, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. 2019. FPGA stream-monitoring of real-time properties. In *ESWEEK-TECS special issue, International Conference on Embedded Software EMSOFT 2019, New York, USA, October 13 - 18*.

[56] Florian-Michael Adolf, Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Christoph Torens. 2017. Stream runtime monitoring on UAS. In *RV 2017* (LNCS). Shuvendu K. Lahiri and Giles Reger, editors. Volume 10548. Springer, 33–49. DOI: 10.1007/978-3-319-67531-2_3.

[57] Peter Faymonville, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. 2017. Real-time stream-based monitoring. *CoRR*, abs/1711.03829. arXiv: 1711.03829. http://arxiv.org/abs/1711.03829.

[58] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. 2016. A stream-based specification language for network monitoring. In *RV 2016* (LNCS). Yliès Falcone and César Sánchez, editors. Volume 10012. Springer, 152–168. ISBN: 978-3-319-46981-2. DOI: 10.1007/978-3-319-46982-9_10.

[59] Jan Baumeister, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. 20019. On the similarities of aircraft and humans: monitoring cps with streamlab. In *CyberCardia Medical CPS Workshop at ESWEEK'19*,

[60] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. 1973. Time bounds for selection. *J. Comput. Syst. Sci.*, 7, 4, 448–461. DOI: 10.1016/S0022-0000(73)80033-9. https://doi.org/10.1016/S0022-0000(73)80033-9.

[61] Lambert Meertens. 1986. Algorithmics : towards programming as a mathematical activity. In *Towards programming as a mathematical activity. Mathematics and computer science*. (January 1986), 289–334.

[62] Author(s) B. P. Welford and B. P. Welford. 1962. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 419–420.

[63] T. F. Chan, G. H. Golub, and R. J. LeVeque. 1982. Updating formulae and a pairwise algorithm for computing sample variances. In *COMPSTAT 1982 5th Symposium held at Toulouse 1982*. H. Caussinus, P. Ettinger, and R. Tomassone, editors. Physica-Verlag HD, Heidelberg, 30–41. ISBN: 978-3-642-51461-6.

[64] Vaughan R. Pratt. 1973. Top down operator precedence. In *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*. Patrick C. Fischer and Jeffrey D. Ullman, editors. ACM Press, 41–51. DOI: 10.1145/512927.512931. https://doi.org/10.1145/512927.512931.

[65] Edsger W. Dijkstra. 1961. Algol 60 translation : An Algol 60 translator for the x1 and Making a translator for Algol 60. Technical report 35. Mathematisch Centrum, Amsterdam. http://www.cs.utexas.edu/users/EWD/MCReps/MR35.PDF.

[66] Klaus Samelson and Friedrich L. Bauer. 1960. Sequential formula translation. *Commun. ACM*, 3, 2, 76–83. DOI: 10.1145/366959.366968. https://doi.org/10.1145/366959.366968.

[67] R. Hindley. 1969. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc*, 146, (December 1969), 29–60.

[68] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. 2016. A stream-based specification language for network monitoring. In *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings* (LNCS). Yliès Falcone and César Sánchez, editors. Volume 10012. Springer, 152–168. ISBN: 978-3-319-46981-2. DOI: 10.1007/978-3-319-46982-9_10. https://doi.org/10.1007/978-3-319-46982-9_10.

[69] Florian-Michael Adolf, Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Christoph Torens. 2017. Stream runtime monitoring on UAS. In *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*, 33–49. DOI: 10.1007/978-3-319-67531-2_3. https://doi.org/10.1007/978-3-319-67531-2_3.

[70] Felice T. Sun, Martha J. Morrell, and Robert E. Wharen. 2008. Responsive cortical stimulation for the treatment of epilepsy. *Neurotherapeutics*, 5, 1, (January 2008), 68–74. ISSN: 1878-7479. DOI: 10.1016/j.nurt.2007.10.069. https://doi.org/10.1016/j.nurt.2007.10.069.

[71] Christianne N. Heck, David King-Stephens, Andrew D. Massey, Dileep R. Nair, Barbara C. Jobst, Gregory L. Barkley, Vicenta Salanova, Andrew J. Cole, Michael C. Smith, Ryder P. Gwinn, Christopher Skidmore, Paul C. Van Ness, Gregory K. Bergey, Yong D. Park, Ian Miller, Eric Geller, Paul A. Rutecki, Richard Zimmerman, David C. Spencer, Alica Goldman, Jonathan C. Edwards, James W. Leiphart, Robert E. Wharen, James Fessler, Nathan B. Fountain, Gregory A. Worrell, Robert E. Gross, Stephan Eisenschenk, Robert B. Duckrow, Lawrence J. Hirsch, Carl Bazil, Cormac A. O'Donovan, Felice T. Sun, Tracy A. Courtney, Cairn G. Seale, and Martha J. Morrell. 2014. Two-year seizure reduction in adults with medically intractable partial onset epilepsy treated with responsive neurostimulation: final results of the rns system pivotal trial. *Epilepsia*, 55, 3, 432–441. DOI: 10.1111/epi.12534. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/epi.12534. https://onlinelibrary.wiley.com/doi/abs/10.1111/epi.12534.

[72] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah. 2019. 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, 421–431. DOI: 10.1007/978-3-030-25540-4_24. https://doi.org/10.1007/978-3-030-25540-4_24.

[73] Pritesh Gandhi, Pradnya Gokhale, Hemant Mehta, and Chinmay Shah. 2013. A comparative study of simple auditory reaction time in blind (congenitally) and sighted subjects. *Indian journal of psychological medicine*, 35, (July 2013), 273–7. DOI: 10.4103/0253-7176.119486.

[74] *12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA*, (2005). IEEE Computer Society. ISBN: 0-7695-2370-6. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=9856.

[75] Christian Colombo and Martin Leucker, editors. *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, volume 11237 of *Lecture Notes in Computer Science*, (2018). Springer. ISBN: 978-3-030-03768-0. DOI: 10.1007/978-3-030-03769-7. https://doi.org/10.1007/978-3-030-03769-7.

[76] Yliès Falcone and César Sánchez, editors. *RV 2016*, volume 10012 of *LNCS*, (2016). Springer. ISBN: 978-3-319-46981-2. DOI: 10.1007/978-3-319-46982-9.

[77] *POPL 1987*, (1987). ACM Press. ISBN: 0-89791-215-2.