Back to the Future: Extending RTLola with Future Offsets

Saarland University

Department of Computer Science

Bachelor's Thesis

submitted by

Eduard Müller

Saarbrücken, December 2025

Supervisor:	Prof. Bernd Finkbeiner, Ph.D.
Advisor:	Jan Baumeister, M.Sc.
	Frederik Scheerer, M.Sc.
Reviewer:	Prof. Bernd Finkbeiner, Ph.D.
	Hazem Torfah, Ph.D.
Submission:	December 03, 2025

Abstract

As software systems become increasingly complex, ensuring correct and efficient runtime behavior is a significant challenge. Runtime monitoring addresses this challenge by verifying the behavior of the program during execution by checking the specifications against the current state of the program. RTLola, a stream-based monitoring language, targets this problem by defining specifications through stream equations on input streams. Output streams compute new values based on both current and past data, while triggers indicate specification violations, allowing an immediate response.
However, we also want to compute values based on future data, so that we can handle mixed temporal properties. This allows us to predict the behavior of the system and respond accordingly, closing the gap between what has happened and what is expected.
The idea of future offsets first came from Lola, a synchronous monitoring language. The move from Lola to the asynchronous approach of RTLola was necessary to support modern systems that require real-time and flexible monitoring. RTLola focuses on efficiently handling current and past data, which is why future offsets are not currently handled.
This thesis proposes an extension to RTLola by integrating the future offset operator, combining the asynchronous and real-time data access features of RTLola with the concepts of future offsets originally introduced in Lola. To demonstrate the applicability of this extension, we revisit the fundamental principles of Lola and present theoretical analysis alongside a practical implementation within the RTLola framework. The evaluation ensures that each set of input streams has a unique evaluation model and that the design prevents computational inefficiencies by keeping the worst-case memory usage constant relative to the trace size.

Acknowledgements

Firstly, I would like to express my gratitude to my supervisor, Prof. Bernd Finkbeiner, for giving me the opportunity to work on this interesting and challenging topic.
In addition, I would also like to thank my advisors, Jan Baumeister and Frederik Scheerer, for their continuous support. I really appreciate the time and effort you invested in our weekly meetings. Your feedback and advice helped me improve my work significantly.
Furthermore, I thank Prof. Bernd Finkbeiner and Dr. Hazem Torfah for reviewing my thesis.
Finally, I would like to thank my family and friends for their encouragement and support throughout the entire process.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne die Beteiligung dritter Personen verfasst habe, und dass ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder singemäß aus Veröffentlichungen oder aus anderweitigen fremden Äußerungen entnommen wurden, sind als solche kenntlich gemacht. Insbesondere bestätige ich hiermit, dass ich bei der Erstellung der nachfolgenden Arbeit mittels künstlicher Intelligenz betriebene Software (z. B. ChatGPT) ausschließlich für folgende zulässige Teilaufgaben: **Code Generation, Text Rewriting/Revision**, und nicht zur Bearbeitung der in der Arbeit aufgeworfenen Fragestellungen zu Hilfe genommen habe. Mir ist bewusst, dass der Verstoß gegen diese Versicherung zum Nichtbestehen der Prüfung bis hin zum Verlust des Prüfungsanspruchs führen kann.

Statement in Lieu of an Oath

I hereby declare that this thesis is my own original work and was completed independently, without unauthorized assistance or unacknowledged sources. Any content derived from publications or other external sources, whether quoted verbatim or paraphrased, has been duly acknowledged and clearly marked as such. I hereby confirm that, in preparing the following thesis, I have used artificial intelligence-based software (such as ChatGPT) solely for the following permitted tasks: **Code Generation, Text Rewriting/Revision**, and not to address the core research questions presented in this thesis. I acknowledge that any breach of this declaration may result in failing the examination and, in severe cases, the right to be examined may be revoked.

Saarbrücken, 03 December, 2025

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, 03 December, 2025

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

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

Saarbrücken, 03 December, 2025

Contents

Chapter 1

Introduction

As software systems become increasingly complex, ensuring correct and efficient runtime behavior is a significant challenge. Runtime monitoring addresses this challenge by verifying the behavior of the program during execution and checking the specifications against the current state of the program. Offline and online monitoring are two approaches to this challenge. Offline monitoring analyzes program traces after execution, helping to detect specification violations during test and simulation phases. In contrast, online monitoring verifies program behavior during execution, using triggers to catch and respond to violations in real time. This dual approach, as implemented in *RTLola* [16], improves both the testing before deployment and the reliability of the operating system.
RTLola [16], a stream-based monitoring language, defines specifications using stream equations. Input streams capture values, while output streams compute new values based on both current and past data. Triggers indicate specification violations and allow online monitoring to respond immediately to violations. Offline monitoring can further support this system by using *RTLola* computations for after-simulation verification. *RTLola*'s prior formulation faced a fundamental limitation in naturally expressing or efficiently verifying complex specifications that inherently require foresight into future states. This created a critical gap, as many practical system requirements demand the ability to anticipate and react to events that "eventually" occur, a concept not natively supported by *RTLola*'s original design.
This thesis proposes an extension to *RTLola* by integrating the future offset operator. Originally introduced in the first version of *Lola*, the future offset operator allows temporal references to future states and extends the expressiveness of the language. Specifications can now define temporal requirements similar to the 'eventually' operator from Linear Temporal Logic (LTL) [37]. This extension improves the ability of *RTLola* to deal with mixed temporal properties and aligns its functionality with the requirements of both offline and online monitoring.

To ensure the practical applicability of this extension, we revisit the fundamental principles of the original *Lola*, such as well-definedness and well-formedness. In particular, we need to ensure that the evaluation has exactly one evaluation model for each set of input streams to avoid ambiguity. This also means that we avoid computational inefficiencies, such as infinity loops of value accesses. We analyze memory bounds to ensure efficient monitorability. This efficiency is similar to previous results where a monitoring strategy was developed [18], that is linear in specification size and the worst-case memory usage of the monitoring algorithm remains constant relative to the trace size. However, when it comes to the online monitoring approach, there is a chance that a value cannot be verified, since the reference value may not exist yet. Here, we introduce new states for output stream values: resolved and unresolved. A value is called unresolved if it depends on a future offset that has not yet occur. When the required future value becomes available, the unresolved value becomes a resolved value. All other values that do not depend on future offsets are by default resolved.
This thesis provides the theoretical foundation for the future offset operator from *Lola* to integrate it into *RTLola*. This includes an extended semantics and new definitions for the formal analysis from *RTLola*. We also present an implementation approach that integrates this operator into the existing *RTLola* toolchain. Finally, we evaluate the memory requirements of the extended *RTLola* through benchmarks, demonstrating its practical behavior and efficiency.

Chapter 2

Related Work

Runtime monitoring is a dynamic analysis tool, which is widely used for verification [22, 26, 13]. This method overcomes challenges such as the state-explosion of model checking, the need for manual invariants in theorem proving, and the restricted behavioral expressiveness of static code analysis [38]. Runtime monitoring employs two main approaches: offline and online monitoring. This dual approach improves both deployment testing and the reliability of operating systems.

Early work in this area relied on temporal logic, to provide automatic monitor synthesis and formal complexity guarantees [30, 25]. Linear temporal logic (LTL) offers a way to express both past and future dependencies within a specification [26, 17]. This enables the direct generation of monitors whose worst-case time and space requirements are known in advance. Here, Finkbeiner and Kuhtz [17] presented a FPGA implementation to capture and monitor this in an hardware-based monitor.

The Temporal Rover [12] is a tool for checking the correctness of software. It creates executable monitors based on LTL specifications for C, C++, Java, Verilog, and VHDL applications. In this case, the LTL specification is written as comments directly in the source code. These specifications are treated as assertions within every program execution cycle. This approach highlights the practical application of formal temporal specifications in real-time verification, a principle that underpins stream-based runtime monitoring languages like *RTLola*.

The Java PathExplorer (JPAX) [23] is a runtime verification tool by NASA, that automatically generates execution events from Java bytecode and checks them against a LTL specification. JPAX can detect temporal property violations and concurrency errors, such as deadlocks and data races, during program execution. This tool exemplifies the automation of event processing and verification against formal temporal properties, demonstrating a need for expressive runtime verification tools that *RTLola* aims to fulfill.

Real-time temporal logics, such as Metric Temporal Logic (MTL) [25], were developed

in response to the shortcomings of LTL in capturing timing constraints across different contexts. Here, MTL extends LTL by adapting temporal operators with explicit numeric time bounds. NASA's R2U2 [33] is a hardware monitoring tool that uses MTL to specify real-time system requirements. The FPGA implementation of R2U2 evaluates MTL formulas through runtime observers and Bayesian networks to determine whether the system satisfies its time-sensitive specifications.

The Signal Temporal Logic (STL) [27] builds on MTL by allowing real-valued signals for monitoring continuous-time system behavior. There exists a FPGA monitor implementation over discrete time, which is based on a subset of STL with past and bounded future operators [24]. In addition to this FPGA monitor, Maler and Nickovic [27] also provided a prototype implementation for checking the properties of simulation traces generated by MATLAB and Simulink [29]. This matrix-based programming provides streaming objects that can be referenced and processed to handle both past and future values in event streams. Simulink Test with Temporal Assessments and Simulink Design Verifier enable formal monitoring and verification of temporal requirements [31].

However, the Boolean verdicts of LTL, MTL and STL are often insufficient, promoting the need for non-binary outcomes. Robust LTL(rLTL) [35, 2, 28] adapts the LTL operators and extends their approach to a multi-valued semantics. This captures different degrees of satisfaction and violation for finite and infinite traces. Nevertheless, when it comes to cyber-physical systems, these logic-based monitoring approaches lack expressiveness. Furthermore, it became necessary to express arithmetic expressions in order to capture the complexity of modern systems. This limitation underscores the necessity for more expressive frameworks.

Over time, stream-based monitoring approaches have been studied to increase expressiveness. Moreover, stream-based specification languages balance the strictness of formal guarantees with the need for rich computational conclusions [37]. In particular, the specifications contain arithmetic expressions alongside past and future offset accesses. This approach comes with a strict memory bound, ensuring formal guarantees and memory efficiency.

Early approaches were based on synchronous languages, which assume that input data arrives simultaneously, synchronizing the input and output data. Lustre [21] is a synchronous dataflow language that specifies the behavior of real-time, reactive systems as equations over infinite input and output streams. Additionally, it provides temporal operators, such as "pre" (previous) and "->" (followed by) for past and future offset accesses.

Similarly, *Lola* [11] is a synchronous, stream-based specification language that allows expressing verification queries as equations over typed input and output streams. Also, both past and future offset operators exist in *Lola*, which sets the stage to express temporal logics. *Lola* has been integrated into an unmanned aircraft systems (UAS) [36, 1].

The first extension of *Lola* was Lola 2.0 [14], which adds new features such as

parametrized streams to support network monitoring. Nevertheless, Lola 2.0 still supports both past and future offset operators.
For more complex modern systems, the synchronous approach became insufficient, which led to an asynchronous monitoring approach. *Lola* has been the basis for the development of many stream-based languages, such as TeSSLa, Striver and *RTLola*.
TeSSLa [9, 10] handles timestamped event streams and includes both past and future offsets [32], as well as recursive operators. TeSSLa specifications compile into bounded-resource monitors which can be run in software or on FPGAs. These monitors have a formally proven correct evaluation algorithm and toolchain support for online and offline verification of time-critical behaviors.
Striver [20, 19] is designed for runtime verification, where each event in a stream has a timestamp, and all timestamps are totally ordered by a global clock. It also introduces the concept of explicit time offsets for both past and future accesses. The language's future offset expressions can therefore compute the next potential timestamp at which an event could occur. However, Striver's current features do not fully support the expression of mixed temporal properties
RTLola [15] builds directly on *Lola*'s ideas but adds parametrized streams, periodic streams, and sliding windows [16, 7, 4]. Moreover, there also exists an integration into an UAS [6, 4]. However, *RTLola* in its current form notably lacks support for future offset accesses.

Chapter 3

Preliminaries

In this chapter, we introduce the necessary background for understanding the concepts and techniques used in this thesis. We start with a brief overview of the *Lola* language in Sect. 3.1, which serves as the foundation for *RTLola*. Then, we introduce *RTLola* itself in Sect. 3.2 and highlight its key features. This sets the stage for the discussions on extending *RTLola* with future offsets.

3.1. Lola

The stream-based specification language *Lola* [11] is designed for monitoring *synchronous systems*, where all stream equations are evaluated in lockstep with a global discrete clock. As seen in Fig. 3.1, *Lola* receives input streams from a system. The input streams are then used to generate output streams, by accessing other streams and evaluating stream equations. At each clock tick, every output stream is computed from the values it depends on at the corresponding timestamps. Triggers are used to generate notifications, if a certain condition is met.

Synchronous System

In the following, we introduce *Lola*'s syntax in Sect. 3.1.1 and its semantics in Sect. 3.1.2, as well as its monitoring algorithm in Sect. 3.1.3.

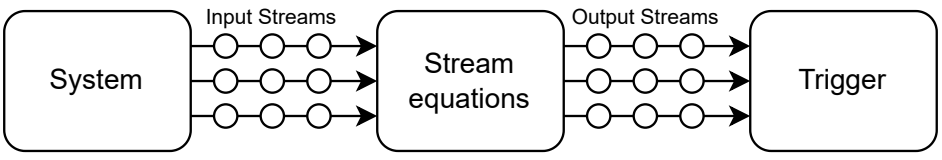


Figure 3.1.: General data flow of monitoring a synchronous system with *Lola*.

3.1.1. Syntax

A *Lola* specification consists of three parts: input streams, output streams, and triggers. Each stream has a type T, while a stream consists of a sequence of values of type T.

Definition 3.1 (Lola Specification [11])

Def. Lola Specification

A Lola *specification* is a tuple $\mathcal{L} = (I, O, \texttt{Trigger})$, where:

- $I = t_1, \ldots, t_m$ is a set of input streams, where each independent stream variable t_i is a stream of type T_i.
- $O = s_1, \ldots, s_n$ is a set of output streams, where each dependent stream variable s_j is a stream of type T_j and is defined by an expression e_j.
- $\texttt{Trigger} = \texttt{trigger}_1, \ldots, \texttt{trigger}_k$ is a set of triggers, where each $\texttt{trigger}_q$ has a boolean expression φ_q.

Additionally, a trigger generates a notification when the boolean expression φ_q evaluates to true.

Based on Def. 3.1, the following specification illustrates the basic syntax of a *Lola* specification:

Example 3.1.1. The specification models a vending machine.

```
input Int itemsSold
input Int itemsRestocked
output Int stock := stock[-1,0] - itemsSold + itemsRestocked
output Bool reorderNeeded := (stock[3,0] < 2) || (itemsSold > itemsSold[-1, 0])
trigger stock < 0
trigger reorderNeeded
```

The input streams `itemsSold` and `itemsRestocked` are getting positive numbers of sold and restocked items respectively. The output stream `stock` updates its state based on the sold and restocked items. Here, both input stream values are accessed, which we call a *synchronous access*. The *offset* expression accesses the previous value of the stock, which is defined by the expression `stock[-1,0]`. The second argument of the offset operator defines a default value, which is used if no previous value exists. Overall the current stock is computed by the previous stock value and the difference of the sold and restocked items. The output stream `reorderNeeded` indicates whether a reorder of items is necessary. This is the case if the stock in three time steps is less than two or if the number of sold items has increased compared to the previous time step. Finally, there are two triggers defined in the specification. The first trigger generates a notification if the stock is less than zero, indicating that there are not enough items available. The second trigger generates a notification if a reorder is needed, as indicated by the output stream `reorderNeeded`. △

Synchronous Access

Offset

In general, an expression $e_j(t_1, \ldots, t_m, s_1, \ldots, s_n)$ of type T_j is used to define the output streams in the *Lola* specification. Besides the expressions used in Example 3.1.1, *Lola* provides several other expressions to define output streams. Therefore, these expressions can consist of constants, input streams, output streams, and operations on these streams, i.e., k-ary operators, *If-Then-Else* expressions, and discrete offsets. The exact definition of these expressions can be found in the original *Lola* paper [11].

3.1.2. Semantic

A *Lola* specification is evaluated over a sequence of discrete time steps, where each time step corresponds to a new value for the input streams. At each time step j, the values of all output streams are computed according to their defining expressions, using the current, previous or future values of the input and output streams. The evaluation proceeds in lockstep, ensuring that all output streams are updated synchronously. This relation is described by an *evaluation model*.

Definition 3.2 (Lola Evaluation Model [11])
An *evaluation model* of a *Lola* specification $\mathcal{L}$ is a tuple $\langle \sigma_1, \ldots, \sigma_n \rangle$. Each stream variable $s_i = e_i(t_1, \ldots, t_m, s_1, \ldots, s_n)$ in $\mathcal{L}$ is mapped to a stream σ_i, which is a sequence of values of length N. The evaluation model must satisfy the following equation for each stream σ_i, its corresponding stream equation e_i and each time step $0 \leqslant j < N$:

Def. Evaluation Model

$$\sigma_i(j) = val(e_i)(j)$$

Here, $val(e_i)(j)$ denotes the value of the expression e_i at time step j. The associated equations of $val(e_i)(j)$ are defined in the original *Lola* paper [11].

According to Def. 3.2, the evaluation model of a *Lola* specification is deterministic and produces an unique output stream for each time step, given a set of input streams. This is ensured by the following property of a *Lola* specification.

Definition 3.3 (Lola Well-Defined [11])
A *Lola* specification $\mathcal{L}$ is called *well-defined* if for every set of typed input streams with length N, there exists a unique evaluation model $\langle \sigma_1, \ldots, \sigma_n \rangle$.

Def. Well-Defined

This means that for every possible input trace, there exists a unique evaluation model that satisfies the semantics of the *Lola* specification. However, there are specifications that are not well-defined.

Example 3.1.2. Consider the following specification:

```
output a := b
output b := a
```

In this case, there exist indefinitely many evaluation models, as long as the values of the streams `a` and `b` are equal. Thus, the specification is not well-defined. Another example is the following specification:

```
output a := !b
output b := a
```

Here, there exists no evaluation model, because the values of the streams `a` and `b` are always opposite to each other. Thus, the specification is also not well-defined. △

These examples showed trivial cases of specifications that are not well-defined. In general, it is undecidable whether a *Lola* specification is well-defined or not for unbounded types [11]. Also for bounded types, it would require an expensive search through all possible evaluation models. Therefore, we need another way to ensure that a *Lola* specification is well-defined.

Further investigations rely on the *dependency graph* of a *Lola* specification, which is defined as follows:

Definition 3.4 (Lola Dependency Graph [11])

Def. Lola Dependency Graph

Let $\mathcal{L}$ be a *Lola* specification. The *dependency graph* of $\mathcal{L}$ is a directed weighted multi-graph $G = \langle V, E \rangle$ where V is the set of streams in $\mathcal{L}$. For each occurrence of a stream variable s_k in the expression defining stream s_i, an edge $e = \langle s_i, \omega, s_k \rangle$ is added to E, where ω is the offset used to access s_k (with $\omega = 0$ for direct accesses).

The dependency graph captures the dependencies between streams in a *Lola* specification, including the type and quantity of stream accesses. For the specification in Example 3.1.1, the dependency graph is illustrated in Fig. 3.2.

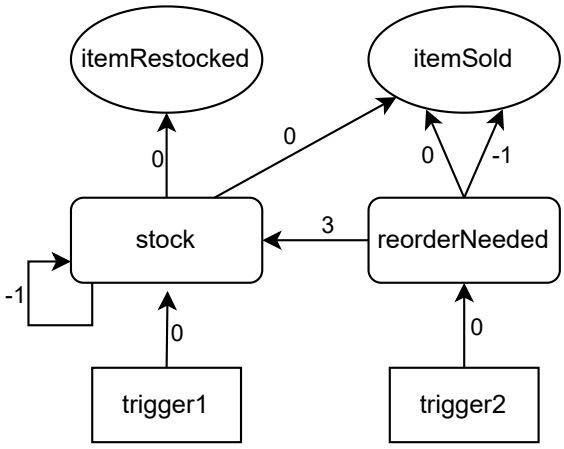


Figure 3.2.: Dependency graph of the *Lola* specification in Example 3.1.1.

(a) Dependency graph from Example 3.1.3 (b) Dependency graph from Example 3.1.4

Figure 3.3.: Examples of dependency graphs.

With this tool we can determine a decidable sufficient condition for a *Lola* specification to be well-defined. This condition is defined as followes:

Definition 3.5 (Lola Well-Formed [11])

A *Lola* specification $\mathcal{L}$ is called *well-formed* if its dependency graph does not contain any cycles with a total weight greater than or equal to zero.

Def. Lola Well-Formed

This property can be checked algorithmically by analyzing the dependency graph for such cycles. For example, in the dependency graph in Fig. 3.2, there is one cycle on the stream `stock`, which has a total weight of -1. Since its total weight is not zero, the specification is well-formed.

Example 3.1.3. Consider the following specification that is not well-formed:

```
output Int a := b[1, 0]
output Int b := a[-1, 0]
```

Here, the dependency graph in Fig. 3.3a contains a cycle with a total weight of zero, as both streams `a` and `b` access each other with an offset of 1 and -1 respectively. This would lead to an infinite loop of value lookups. Thus, the specification is not well-formed. △

In general, whenever there are no zero-weight cycles, every stream equation can be evaluated in a unique and deterministic way for any input trace. Therefore, a well-formed *Lola* specification is also well-defined, but the converse is not necessarily true.

Example 3.1.4. Consider the following specification:

```
output Bool a := a && !a
```

In this case, the dependency graph in Fig. 3.3b contains a cycle with a total weight of zero, as the stream `a` accesses itself with an offset of 0. Thus, the specification is not well-formed. However, the specification is well-defined, as the stream `a` evaluates always to false. △

We call a specification *well-analyzed*, if it is both, *well-defined* and *well-formed*. A *well-analyzed* specification forms the basis for the monitoring algorithm, which is described in the following section.

Well-Analyzed

3.1.3. Monitoring Algorithm

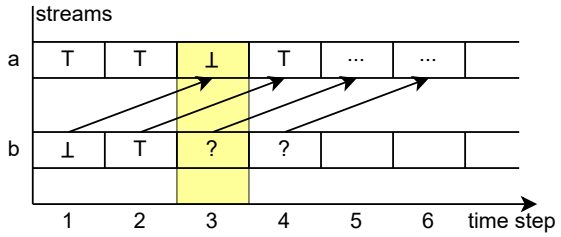
There are two monitoring approaches with a *well-analyzed Lola* specification: offline and online monitoring. In the offline monitoring approach, the completed trace of a system's run is stored, which will be analyzed afterwards. Here no further adjustments are needed, as all operators are bounded by the trace length. However, the online monitoring approach analyzes the system's trace on the fly, which lead to unbounded offset operators. This means that at the moment we want to access a future value, the value may not exist yet.

(un)resolved equations

Lola's online monitoring algorithm provides a stalling mechanism to address this problem. It divides the stream equations into *resolved and unresolved equations* [11], i.e. all stream equations that depent on values in the future are stored as unresolved equations. After the corresponding values occur, the values of those equations are stored as resolved equations. The following example illustrates this mechanism:

Example 3.1.5. Consider the following specification and its corresponding trace.

```
input Bool a
output Bool b := a[2, false]
```



Here, the output stream `b` accesses the input stream `a` with an offset of 2. This means that at some time step j, the value of `b` depends on the value of `a` at time step $j + 2$. At time step 3, the equation for `b` is unresolved, as the value of `a` at time step 5 does not exist yet. Then, the value of `b` is marked with the `?` symbol. However, at time step 2, the value of `b` is resolved, as the value of `a` at time step 4 exists. Thus, the value of `b` is computed and stored. If the trace ends and `b` wants to access a value of `a` beyond the end of the trace, the default value is used. △

With this mechanism, *Lola* can delay the evaluation of values that haven't occured yet. Furthermore, *Lola* must guarantee that only a finite number of unresolved equations are present at any time. This is necessary to ensure that the memory consumption of the monitoring algorithm is bounded. These kinds of specifications are called *efficiently monitorable specifications* and are defined as follows:

Definition 3.6 (Efficiently Monitorable Specifications [11])

A *Lola* specification is called *efficiently monitorable*, if there is a trace length independent and strict memory bound on the number of unresolved equations during the online monitoring.

Def. Efficiently Monitorable

This can be archived if the dependency graph of the *Lola* specification does not contain positive cycles, i.e., cycles with total weight greater than zero. With this condition, the monitoring algorithm can ensure that the number of unresolved equations is bounded by the maximum offset per equation used in the specification.

For example the specification in Example 3.1.1 is efficiently monitorable, as its dependency graph in Fig. 3.2 does not contain any positive cycles. However, consider the following specification and its dependency graph:

```
output Int a := a[1, 0] + 1
```

Here, the dependency graph contains a positive cycle on the stream `a` with a total weight of 1. This means that at each time step j, the value of `a` depends on the value of `a` at time step $j + 1$. Thus, the number of unresolved equations grows unbounded over time, as the value of `a` can never be resolved and the end of the trace is unknown. Therefore, the specification is not efficiently monitorable.

3.2. RTLola

In this section, we introduce the stream-based specification language *RTLola* [15], on which the subsequent chapters build. *RTLola* is an extension of *Lola* [11], which allows the monitoring of real-time systems. It introduces concepts to support *asynchronous systems*, where the input streams are ticking at different times. As seen in Fig. 3.5, *RTLola* receives input streams from an system. The input streams are then used to generate output streams, by accessing other streams and evaluating stream equations. Triggers are used to generate notifications, if a certain condition is met. However, the evaluation of the streams may not be synchronized by a global discrete clock. Instead, each stream can have its own clock, which is more capable for real-time systems.

Asynchronous System

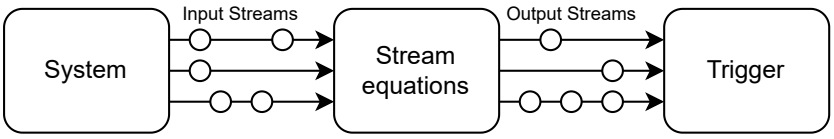


Figure 3.5.: General data flow of monitoring a real-time system with *RTLola*.

The following specification illustrates the basic syntax of a *RTLola* specification:

Example 3.2.1. The following specification is a translation of the vending machine example from Example 3.1.1 into *RTLola*. As *RTLola* does not support future offsets, we ignore the output stream `reorderNeeded` and its corresponding trigger.

```
input itemsSold: Int
input itemsRestocked: Int
output stock: Int := stock.offset(by: -1).defaults(to: 0) - itemsSold + itemsRestocked
trigger stock < 0 "out of stock"
```

The input streams track the number of sold and restocked items respectively. The output stream `stock` updates its state based on its own previous value and the difference of the input streams. Here, we use the *offset* expression to access the previous value of the stock, which is defined by the expression `stock.offset(by: -1)`. The second argument of the `defaults(to: 0)` operator defines a default value, which is used if no previous value exists. The input streams are accessed directly, which we call a *synchronous access*. The trigger is used to generate a notification, if the stock is less than zero. In this case, the notification is "out of stock", s.t. the owner of the online shop can react to this situation. △

Offset

Synchronous Access

Value Types

In *RTLola*, each stream has a *value type*, which defines the type of the values that the stream can hold. It can be any common type, such as `Int`, `Float`, or `Bool`. For instance, in Example 3.2.1, the input streams `itemsSold` and `itemsRestocked` have the value type `Int`. This means that the specification expects integer values for these streams. The input streams need to specify their value type. The output stream `stock` can either infer its value type based on the expressions used in the stream equation or it can be explicitly annotated. In this case, the value type of `stock` is explicitly annotated as `Int`. Triggers do not have an explicit annotated value type, as they always have boolean value types.

Value Type

Pacing Type

As *RTLola* is designed for monitoring real-time systems, it may not be the case that all input streams are ticking at the same time. This means that the input streams can tick asynchronously. Also there is a new feature to define output streams, which can be evaluated periodically. In general, *RTLola* does not have a global discrete clock, which would synchronize the streams. Instead, each stream can have a local discrete clock, which is used to pace the evaluation of the stream equations. The local discrete clock is defined by *activation condition* of a stream equation, which specifies when the output stream is evaluated.

Activation Condition

In Example 3.2.1, the output stream `stock` does not have an explicit activation condition. In this case, the activation condition is inferred based on the accessed input streams.

Here, both input streams `itemsSold` and `itemsRestocked` are accessed synchronously. This means that the output stream `stock` is evaluated whenever both input streams are ticking.

However, the activation condition can also be explicitly defined, which is done by the `@` symbol. This activation condidition consists of a *pacing type*, which can be either periodic or event-based, as described below.

Pacing Type

- **Periodic Pacing:** This type of pacing consists of a frequency, which can be any value with an unit `Hz` or `s`. The output stream is evaluated periodically based on the given frequency. In Example 3.2.1, we could want to evaluate the `stock` stream once every second:

  Periodic Pacing

  ```
  output stock : Int @1Hz := stock.offset(by: -1).defaults(to: 0)
                           - itemsSold.hold(or: 0) + itemsRestocked.hold(or: 0)
  ```

  Since the input streams do not tick at a specific frequency, we cannot access the input streams synchronously. Thus, there is an *asynchronous access* to the input streams. Here, we use the `hold(or: 0)` operator to access the last seen value of the input streams. If there is no previous value, it returns the default value, which is zero in this case.

  Asynchronous Access

- **Event-Based Pacing:** This type of pacing consists of a positve boolean expression over input streams. Whenever an input stream ticks, it corresponds to `true` in this expression and `false` otherwise. The output stream is evaluated whenever the boolean expression evaluates to `true`.
  In Example 3.2.1, we could explicitly annotate the activation condition:

  Event-Based Pacing

  ```
  output stock : Int @(itemsSold && itemsRestocked) :=
     stock.offset(by: -1).defaults(to: 0) - itemsSold + itemsRestocked
  ```

  This is equivalent to the inferred activation condition. However, the activation can also depend only on one input stream:

  ```
  output stock : Int @itemsSold :=
     stock.offset(by: -1).defaults(to: 0) - itemsSold + itemsRestocked.hold(or: 0)
  ```

  Here, the stream `stock` will be evaluated whenever the input stream `itemsSold` ticks. In this case, there is no gaurantee that both input streams are ticking at the same time, s.t. the input stream `itemsRestocked` cannot be accessed synchronously. Therefore, we use the `hold(or: 0)` operator to access the last seen value of the input stream `itemsRestocked`. If there is no previous value, it returns the default value, which is zero in this case.

Note that by explicitly annotating the activation condition, we can synchronize multiple output streams. In this cases, a decision must be made as to whether synchronous or asynchronous access can be used. This also means that the discrete clocks of the output streams may synchronize under certain conditions, i.e., if the activation condition is the same for multiple output streams. In this case, the output streams are evaluated at the same time, which means that they can access each other synchronously.

Sliding Window

Sliding Window

A periodic pacing can also be used for aggregations over a time window. This is called a *sliding window*. The following example is an extention of the Example 3.2.1:

```
output order_count : UInt @1Hz := itemsSold.aggregate(over: 1s, using: count)
```

The output stream `order_count` evaluates once every second. The operator `aggregate(over: 1s, using: count)` is used to compute the number of orders in the last second. This is specified by the keyword `count` in this operator. Note that the input stream `itemsSold` is accessed asynchronously, as it may not tick at the same time as the output stream `order_count`.

Semantic Filter

In Example 3.2.1 we saw a simple way to define output streams. However, *RTLola* also provides a more general way to define output streams by using the `eval` keyword. Therefore we can rewrite the output stream `stock` from Example 3.2.1 as follows:

```
output stock : Int
   eval @(itemsSold && itemsRestocked) when true
        with stock.offset(by: -1).defaults(to: 0) - itemsSold + itemsRestocked
```

Semantic Filter

Here, the `eval` keyword is used to define the evaluation of the output stream `stock`, followed by the activation condition `@(itemsSold && itemsRestocked)`. Additionally, the activation condition can have a *semantic filter*, which is defined by the `when` keyword. The semantic filter is a boolean expression that must be true for the output stream to be evaluated. In this case, the semantic filter is `true`, which means that the output stream is always evaluated when the activation condition is met. Finally, the `with` keyword is used to define the expression that computes the value of the output stream. This is the same as before.

The semantic filter can be used to further restrict the evaluation of the output stream. For example, we could define an output stream that only evaluates if an large number of items are sold and then counts the number of large orders:

```
output large_orders : Int
   eval @itemsSold when itemsSold > 10
        with large_orders.offset(by: -1).defaults(to: 0) + 1
trigger large_orders > 5 "high number of large orders"
```

Here, the trigger notifies the owner, if the specified amount of large orders is exceeded. In this case, the notification "high number of large orders" is generated.

Remark 3.2.1. *In this section, we only considered some features of the* RTLola *language. Features like `spawn`-clauses, `close`-clauses and parametrization are not considered here, since they are not taken into account in this thesis. For further information on the* RTLola *language, and features like parametrization, we refer to the tutorial by Baumeister et al. [4].*

3.2.1. Semantic

In this subsection, we define the *evaluation model* of a *RTLola* specification, which is used to define the semantics of the specification.

Definition 3.7 (Evaluation Model [8])

An *evaluation model* $\mathbb{W}$ of a *RTLola* specification is a combination of a $\mathtt{StreamMap}$ and a $\mathtt{TimeMap}$. It is a collection of timed data streams over a set of input streams $\mathrm{ID}^{\uparrow}$ and output streams $\mathrm{ID}^{\downarrow}$. The $\mathtt{StreamMap}$ maps each stream reference $\mathrm{ID}^{*} = \mathrm{ID}^{\uparrow} \uplus \mathrm{ID}^{\downarrow}$ to a stream, which is a function from time points to values. The $\mathtt{TimeMap}$ maps each time point to its corresponding real-time value. Formally, we define:

Def. Evaluation Model

$$\begin{aligned} \mathtt{Stream} &:= \mathtt{Time} \to \mathbb{V}_{\bot} \\ \mathtt{StreamMap} &:= \mathrm{ID}^{*} \to \mathtt{Stream} \\ \mathtt{TimeMap} &:= \mathtt{Time} \to \mathbb{R} \\ \mathbb{W} &:= \mathtt{StreamMap} \times \mathtt{TimeMap} \end{aligned}$$

Here, $\mathbb{V}_{\bot}$ is the set of all possible values, including the undefined value $\bot$. The $\mathtt{Time}$ is the set of all discrete time points.

Based on this definition, we can define the semantics of a *RTLola* specification. We want to denote a real-time timestamp at a discrete timestamp $t \in \mathtt{Time}$ by $\omega(t)$. The value of a stream reference $\mathtt{sid}$ at time t can be accessed by $\omega(\mathtt{sid})(t)$. In case the pacing or semantic filter prevents the evaluation of this stream at time t, the value is undefined and denoted by $\bot$.

Definition 3.8 (Semantics [3])

The *semantics* of a *RTLola* specification φ with the evaluation model $\mathbb{W}$ is defined as the set:

Def. Semantics

$$\begin{aligned} [\![\varphi]\!] = \{\omega \in \mathbb{W} \mid \forall \mathtt{sid} \in \mathrm{ID}^{\uparrow}. \forall t \in \mathtt{Time}. \varphi(\mathtt{sid}) \Downarrow_{\omega}^{t} \omega(\mathtt{sid})(t) \\ \wedge \forall t \in \mathtt{Time}. \omega(t) < \omega(t+1)\} \end{aligned}$$

Here, $\varphi(\mathtt{sid}) \Downarrow_{\omega}^{t} v$ means that the expression $\varphi(\mathtt{sid})$ evaluates to the value v at time t in the evaluation model ω. The second condition ensures that the real-time timestamps are strictly increasing.

According to the semantics, each output stream value in the evaluation model is calculated correctly based on the defining equations of the specification. The corresponding inference rules for the evaluation of expressions are listed in Sect. A.1.1 as well as their corresponding functions in Sect. A.2.1.

Similar as in Def. 3.3 in *Lola*, we define the *well-defined* property for *RTLola* specifications.

→ Def. 3.3, p. 9

Definition 3.9 (RTLola Well-Defined [3])

Def. RTLola Well-Defined

A *RTLola* specification φ is called *well-defined*, if every possible input trace $I \in \mathit{Time} \to \mathit{InputValues}$ with $\mathit{InputValues} : \mathrm{ID}^{\uparrow} \to \mathbb{V}$, there exists an unique evaluation model $\omega \in [\![\varphi]\!]$ with $\forall t \in \mathit{Time}.\forall i \in \mathrm{ID}^{\uparrow}.\omega(i)(t) = I(t)(i)$.

According to this definition, a *RTLola* specification is well-defined if for every possible input trace, there exists only one evaluation model that satisfies the semantics of the specification. However, as in *Lola*, there are specifications that are not well-defined, as seen in Example 3.1.2. As in *Lola*, it is undecidable whether a *RTLola* specification is well-defined or not, since it would require an expensive search through all possible evaluation models. Therefore, there are multiple analysis tools to ensure this property, which are described in the following section.

3.2.2. Analysis

In this section, we introduce the different analysis stages of a *RTLola* specification. The analysis stages are necessary to guarantee that the monitoring algorithm can be applied to the specification.

Dependency Analysis

To ensure that a *RTLola* specification is well-defined, we can use the *dependency graph* of the specification. The dependency graph forms the basis for our analysis, i.e. it describes the relation between streams. This means, it contains information about the type and quantity of stream accesses. However, the dependency graph of *Lola* in Def. 3.4 is not sufficient for *RTLola*, as it does not consider the new features of *RTLola*, such as asynchronous access and sliding windows. Therefore, we define the dependency graph as follows:

→ Def. 3.4, p. 10

Definition 3.10 (RTLola Dependency Graph [34])

Def. RTLola Dependency Graph

The *dependency graph* of a *RTLola* specification Φ is a directed weighted multi-graph $D_\Phi = \langle V, E \rangle$, where V is the set of streams references ID^* in Φ. Each edge represents a dependency between two streams, where $E \subseteq V \times L_D \times V$. The dependency label L_D consists all kinds of dependencies, i.e.:

$$L_D = \{\mathit{Filter}, \mathit{Eval}\} \times (\{\mathit{Sync}\} \cup \{\mathit{Hold}\} \cup (\{\mathit{Offset}\} \times \mathbb{Z}) \cup (\{\mathit{Aggregation}\} \times \mathcal{F} \times \mathbb{R}))$$

Here, the fist component describes whether the dependency is located in the semantic filter or in the evaluation expression. The second component describes the type of access. $\mathcal{F}$ describes the set of all aggregation functions. The edge generation is defined as follows:

- Input stream references $\mathrm{ID}^{\uparrow}$ do not have outgoing edges.

- For an output stream reference $o \in ID^{\downarrow}$, for each accessed stream reference $o' \in ID^{*}$:
  - in the semantic filter of o, there is an edge $(o, (\mathtt{Filter}, l), o')$, where l is the type of access used to access o'.
  - in the evaluation expression of o, there is an edge $(o, (\mathtt{Eval}, l), o')$, where l is the type of access used to access o'.
  - if o' is accessed multiple times in the semantic filter or evaluation expression of o, multiple edges are added.
- For an edge $(o, (t, l), o')$ of type t, if the access type l is:
  - an offset expression, l is annotated as $(\mathtt{Offset}, n)$, where n is the offset used in the expression.
  - an aggregation expression, l is annotated as $(\mathtt{Aggregation}, f, r)$, where f is the aggregation function and r is the real-time window used in the aggregation.
  - a synchronous or hold expression, l is annotated as $\mathtt{Sync}$ or $\mathtt{Hold}$ respectively.

The resulting edges are added to E.

The dependency graph is used to represent the dependencies between the streams in the *RTLola* specification. Each stream is represented as a vertex in the graph, and each dependency between streams is represented as a directed edge. The edges are labeled with information about the type of access and whether the dependency is located in the semantic filter or in the evaluation expression. The weight of an edge is determined by the type of access used to access the stream. The following example illustrates the dependency graph of a *RTLola* specification:

Example 3.2.2. Consider the *RTLola* specification from Example 3.2.1, but with all introduced features:

```
input itemsSold: Int
input itemsRestocked: Int
output stock: Int @itemsSold := stock.offset(by: -1).defaults(to: 0)
                        - itemsSold + itemsRestocked.hold(or: 0)
output order_count : UInt @1Hz := itemsSold.aggregate(over: 1s, using: count)
output large_orders : Int
   eval @itemsSold when itemsSold > 10
   with large_orders.offset(by: -1).defaults(to: 0) + 1
trigger large_orders > 5 "high number of large orders"
trigger stock < 0 "out of stock"
```

The dependency graph is illustrated in Fig. 3.6 △

Based on the dependency graph, we can define the *well-formed* property for *RTLola* specifications. Just as in Def. 3.5 for *Lola* specifications, we consider the cycles in the → Def. 3.5, p. 11

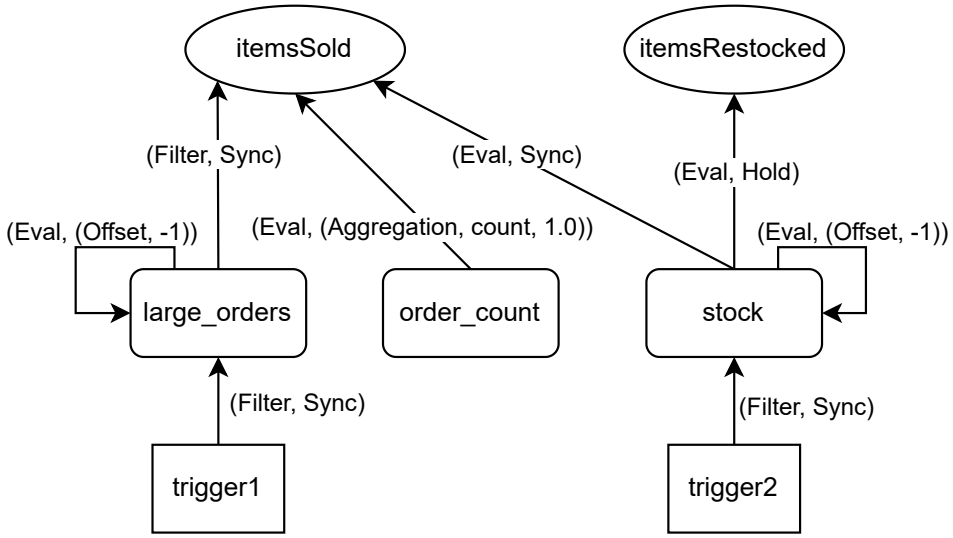


Figure 3.6.: Dependency graph of the *RTLola* specification in Example 3.2.2.

dependency graph. However, *RTLola* introduces new types of accesses. Here, we only consider the synchronous access with a weight of zero and the offset access with a weight of the offset. All other types of access, i.e. hold and aggregation, are considered to have a zero weight. This is because hold accesses the last seen value, which is always in the past, and aggregation accesses a time window in the past. Therefore, they can be handled as past value access.

Definition 3.11 (RTLola Well-Formed [4, 34])

Def. RTLola Well-Formed

Given a *RTLola* specification Φ and its corresponding dependency graph $D_\Phi = \langle V, E \rangle$. The *RTLola* specification Φ is called *well-formed*, if the dependency graph D_Φ does not contain any cycles:

- with the accumulated edge weight of zero.
- that contain an `Filter`-edge and an `Offset`-edge:

$$\forall\{(o_1,(t_1,l_1),o_2),\ldots,(o_n,(t_n,l_n),o_1)\} \subseteq E : \exists i : l_i = (\texttt{Offset},\_) \wedge \forall i : t_i \neq \texttt{Filter}$$

This holds under the assumption, that for each edge $(o,(t,l),o') \in E$, both streams have either periodic or event-based pacings.

This ensures that there are no circular dependencies between the streams, which would lead to an infinite loop of value lookups.
In the dependency graph in Fig. 3.6, there are two cycles, one on the stream `stock` and one on the stream `large_orders`. However, both cycles have a total weight of -1, as both

streams access themselves with an offset of -1. Thus, the specification is well-formed. In Example 3.1.3, we saw an example of a *Lola* specification that is not well-formed. However, this specification uses future offsets, which are not supported in *RTLola*. Since all operators in *RTLola* are accessing the past, it is only possible to have cycles with total weight zero with synchronous accesses. Thus, a *RTLola* specification is not well-formed if there exists a cycle in the dependency graph with only non-offset accesses. Just as in *Lola*, a well-formed specification also implies a well-defined specification. But the converse is not necessarily true as seen in Example 3.1.4.

Order Analysis

Order analysis determines the sequence in which streams are evaluated during monitoring. This is essential for ensuring that all dependencies are resolved before a stream is computed, especially in the presence of offsets. The evaluation order is derived from the dependency graph by determining a partial order among streams. Here we must guarantee that input streams are processed first, followed by output streams according to their dependencies. All definitions are introduced and proven by Schwenger [34]. The first property is called *evaluation order*, which is defined as follows:

Definition 3.12 (Evaluation Order [34])

Given a well-formed *RTLola* specification Φ and its corresponding dependency graph $D_\Phi = \langle V, E \rangle$. The *evaluation Order* $\prec \subseteq \mathrm{ID}^* \times \mathrm{ID}^*$ is the partial order on streams, which must satisfy the following criteria:

Def. Evaluation Order

1. $\forall in \in \mathrm{ID}^{\uparrow} : \forall out \in \mathrm{ID}^{\downarrow} : in \prec out$
2. $\forall (o, (t, l), o') \in E : l \neq (\mathit{Offset}, \_) \implies o' \prec o$
3. $\forall (o, (t_1, l_1), o') \in E, \forall (o', (t_2, l_2), o'') \in E :$

$$((t_1, l_1) = (\mathit{Eval}, (\mathit{Offset}, \_))) \wedge ((t_2, l_2) = (\mathit{Filter}, \_)) \implies o'' \prec o$$

Triggers are evaluated after all streams, as they are not accessed by any other streams.

To get a better feeling on how those criteria apply on a *RTLola* specification, we will now take a look at the following examples:

Example 3.2.3. Consider the following *RTLola* specification:

```
input a: Int
output b: Int := c + a.offset(by: -1).defaults(to: 0)
output c: Int := b.offset(by: -1).defaults(to: 0) + 1
```

Now we want to analyze the evaluation order of this specification based on the dependency graph in Fig. 3.7a. First, we know that `a` will be evaluated first, because it is an

input stream. Second, there is the output stream `b`. It accesses stream `c` synchronously. Hence, `c` must be evaluated before `b`, i.e. $c \prec b$. Also, `b` accesses `a` with an offset expression. Since `a` is an input stream, it holds $a \prec b$. Third, there is the output stream `c`, which access `b` with an offset expression. Therefore, there will be no additional rule added to the evaluation order.

Overall we get the following evaluation order: $a \prec c \prec b$. △

This example illustrates, how the first and second criterion is applied. Here, we check every edge in the dependency graph and try to apply these criteria. If we can apply it, the rule is added to the partial order $\prec$.

Now we take a look at the third criterion:

Example 3.2.4. Consider the following *RTLola* specification:

```
input a: Bool
output b eval when d with a
output c eval when d with b.offset(by: -1).defaults(to: false)
output d @a := a
```

Now, we analyze the evaluation order based on the dependency graph in Fig. 3.7b. The input stream `a` will be evaluated first. The output stream `b` has a synchronous `Filter`-edge to stream `d`, s.t. $d \prec b$. Also, there is a synchronous `Eval`-edge from `b` to `a`, s.t. $a \prec b$. The output stream `c` has also a `Filter`-edge to stream `d`, s.t. $d \prec c$. Based on the `Eval`-edge with an offset expression from `c` to `b`, we can apply the third criterion. It also proves that $d \prec c$. Lastly, the output stream `d` accesses synchronously `a`, s.t. $a \prec d$.

Overall, we get: $a \prec d$, $d \prec c$ and $d \prec b$. △

This criterion particulary ensures that semantic filter conditions are resolved before that stream is evaluated.

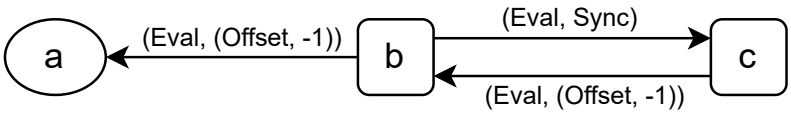
The evaluation order is a side-product of the proof for well-definedness, which ensures that an *RTLola* specification has an unique evaluation order. It is constructed by performing a topological sort on the dependency graph, respecting the criteria defined above. This guarantees that each stream is evaluated only after all its dependencies have been resolved. As a result, the monitoring algorithm can process streams in the computed order, ensuring correctness and determinism for well-formed *RTLola* specifications. However, the *RTLola* compiler needs to know exactly when to evaluate which stream. Here, multiple streams can also be evaluated concurrently, if they do not depend on each other, which cannot be represented by the evaluation order. Therefore, we define the evaluation layer as follows:

Definition 3.13 (Evaluation Layer [34])

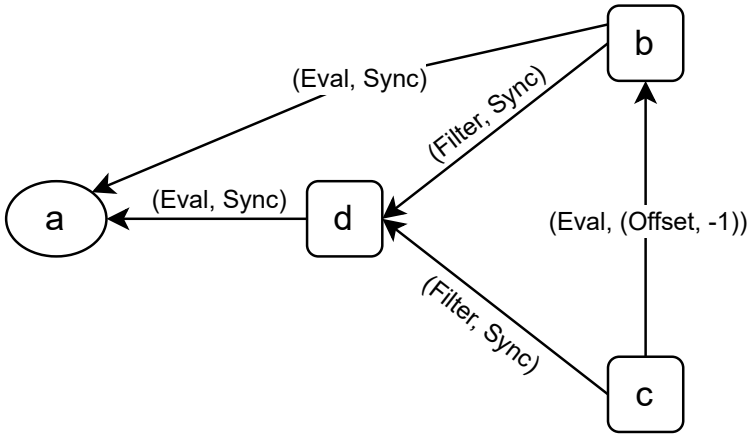
Def. Evaluation Layer

Let $\prec$ be the evaluation order of a well-formed *RTLola* specification Φ. The *evaluation layer* $\mathtt{Layer}_{\prec}(s)$ for $s \in \mathrm{ID}^*$ is computed according the following criteria:

1. $\forall in \in \mathrm{ID}^{\uparrow} : \mathtt{Layer}_{\prec}(in) = 0$



(a) Dependency graph of Example 3.2.3



(b) Dependency graph of Example 3.2.4

Figure 3.7.: Dependency graphs for the evaluation order examples.

2. $\forall o \in ID^{\downarrow} : Layer_{\prec}(o) = 1 + \max_{o' \in ID^*} \{Layer_{\prec}(o') \mid o' \prec o\}$

The greatest evaluation layer is denoted as $\lambda^* = \max_{o \in ID^*} \{Layer_{\prec}(o)\}$.

This property organizes streams into distinct levels based on the evaluation order. Therefore, input streams are in Layer 0, since they must be resolved first. All other streams are placed in their respective layers based on the calculated evaluation order $\prec$.

For Example 3.2.3, we have the evaluation order $a \prec c \prec b$. Therefore we get the following evaluation layers:

$$Layer_{\prec}(a) = 0$$

$$Layer_{\prec}(c) = 1$$

$$Layer_{\prec}(b) = 2$$

Here, the streams have a strict order to ensure an unique evaluation. Therefore b must wait for c to be evaluated, which must wait for a to be evaluated. However in Example 3.2.4, we have the evaluation orders $a \prec d$, $d \prec c$ and $d \prec b$. Therefore we get the following evaluation layers:

$$Layer_{\prec}(a) = 0$$

$$Layer_{\prec}(d) = 1$$

$$Layer_{\prec}(b) = Layer_{\prec}(c) = 2$$

This means, that the streams `b` and `c` can be evaluated concurrently without losing an unique evaluation. Then, both streams must wait for `d` to be evaluated, which must wait for `a` to be evaluated. Therefore, both specifications are well-defined, while ensuring an unique evaluation order.

Memory Analysis

The memory bound analysis determines the amount of values that must be stored per stream. This analysis is crucial for ensuring that the monitoring algorithm can operate within finite memory constraints. Specifically, it examines the dependency graph to identify the maximum offset required for each stream, which directly translates to the number of past values that must be retained in memory during monitoring. Streams with offsets will require more memory, while streams with synchronous or asynchronous accesses will have lower memory requirements.

Definition 3.14 (Stream Memory Bound [34])

Def. Stream Memory Bound

Given a *RTLola* Specification Φ and its corresponding dependency graph $D_\Phi = \langle V, E \rangle$. The *stream memory bound* $\mu(o)$ of a stream o is defined as:

$$\mu(o) = \max\left\{1, 1 + \max_{e \in E}\{\,|n| \mid e = (\_, (\mathrm{Offset}, n), o)\}\right\}$$

If a stream is accessed by an offset, we need to save the past n values and additional the current value.

This means that the memory bound of a stream is determined by the maximum offset of the edges in the dependency graph that point to the stream. The result of this analysis is a per-stream memory bound, which guarantees that the monitoring process remains efficient and scalable.
This behavior is visible in the following example:

Example 3.2.5. Consider the following *RTLola* specification:

```
input a: Bool
output b := a.offset(by: -2).defaults(to: false)
output c := b.offset(by: -4).defaults(to: false)
output d := c
```

The corresponding dependency graph is shown in Fig. 3.8. Therefore the stream memory bounds are:

σ	a	b	c	d
$\mu(\sigma)$	3	5	1	1

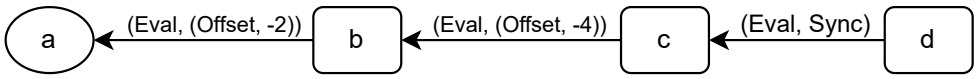


Figure 3.8.: Dependency graph of Example 3.2.5

△

Particulary, if a stream is accessed by an offset expression, the stream needs to store at least the amount of past values required by the largest negative offset used to access it, plus one for the current value. For example, if a stream is accessed with an offset of $-n$, it must retain $n + 1$ values in memory. If there are no offset accesses, the stream only needs to store its current value, resulting in a memory bound of 1.

3.2.3. Framework

The *RTLola* framework consists of several components, as illustrated in Fig. 3.9. These components can be divided into two main parts: the frontend and the backend. The frontend is responsible for parsing and analyzing the *RTLola* specification. Here, the specification is parsed and checked by the generation of an abstract syntax tree (AST).

After this grammar check, the AST is analyzed in several stages, i.e. different analysis modes are applied to the AST. The analysis modes are described below:

1. **Base Mode:** This mode represents the initial state of the AST after parsing. It contains all the information from the specification, but no analysis has been performed yet.

2. **Type Mode:** This mode checks the types of the streams and expressions in the specification. It ensures that the types are consistent and that there are no type errors.

3. **Dependency Analysis Mode:** This mode analyzes the dependencies between the streams in the specification. It constructs the dependency graph and checks for well-formedness.

4. **Ordered Mode:** This mode determines the evaluation order of the streams based on their dependencies. It ensures that the streams are evaluated in the correct order.

5. **Memory Bound Mode:** This mode calculates the memory bounds for the streams based on their dependencies and offsets. It ensures that a memory bound can be determined for each stream.

6. **Complete Mode:** This mode represents the final stage after all analysis modes have been applied. It contains all the information from the specification.

By passing these stages, a high-level intermediate representation (HIR) of the specification is generated.

Afterwards, the HIR runs through a lowering process, which transforms the HIR into a mid-level intermediate representation (MIR). This process simplifies the specification and prepares it for code generation. The MIR is then used to generate executable code for monitoring the specified system. The backend is responsible for this code generation.

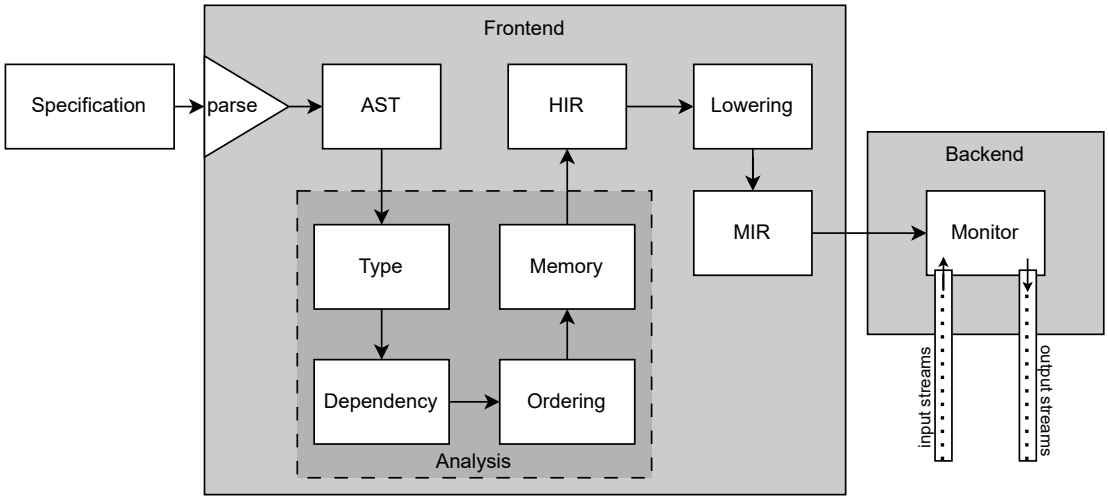


Figure 3.9.: Overview of the *RTLola* framework.

Chapter 4

The Future Offset in Theory

In this chapter, we introduce *RTLola+*, which incorporates the future offset operator to the base language *RTLola*. We begin in Sect. 4.1 with an overview of *RTLola+*, motivating the need for the future offset operator through practical examples. Sect. 4.2 extends the formal semantics of *RTLola+* to accommodate this new feature. Afterwards we analyze in Sect. 4.3 the implications of future offsets on the dependency analysis. In Sect. 4.4, we take a look at the evaluation order of specifications with future offsets. Finally, in Sect. 4.5, we investigate the memory requirements introduced by future offsets. We establish conditions under which specifications can be monitored efficiently, ensuring that memory usage remains manageable.

4.1. Overview

The *future offset* operator enables references to stream values that will be available at a later point in time. This feature enhances the expressiveness of specifications beyond the capabilities of the baseline language *RTLola*. A typical application is the introduction of an artificial delay in the evaluation of a stream.

Example 4.1.1. Consider a monitoring system in a smart home. Here, we monitor the current number of people that are currently in a room and the current light setting. The desired behavior is to notify the system whenever the lights are on while there will be no people in this room. This behavior can be expressed concisely using future offsets:

```
input num_people : Int
input lights_on : Bool
trigger lights_on.hold(or: false) &&
         (num_people.offset(by: 1).defaults(to: 0) == 0) &&
         (num_people.offset(by: 2).defaults(to: 0) == 0)
```

In this specification, the trigger checks the last seen light setting and delay the evaluation by two time steps to see whether there will be no people in the room. If the lights are on

and there will be no people in the room in the next two time steps, the trigger notifies the system after this delay. The future offset operator allows us to express this delay directly and intuitively. Also the pacing plays a significant role here, as the input streams may arrive at different rates, which makes a synchronous evaluation more complicated. In this case, the trigger evaluates whenever the input stream `num_people` delivers a new value. △

Besides introducing delays, future offsets also enable reasoning about future values in a more general sense. For instance, temporal operators from Linear Temporal Logic (LTL) can be expressed using future offsets.

Example 4.1.2. Consider a monitoring system for an unmanned drone. The drone produces two input streams: `takeoff`, which becomes true when the drone takes off, and `altitude_reached`, which becomes true once the desired altitude is reached. We want to specify that whenever the drone takes off, it eventually reaches the desired altitude. In LTL, this can be expressed as the formula φ:

$$\varphi \models \square(\mathtt{takeoff} \rightarrow \Diamond \mathtt{altitude\_reached})$$

This property can be encoded in RTLola using future offsets as follows:

```
input takeoff: Bool
input altitude_reached: Bool
output eventually_altitude_reached @altitude_reached := altitude_reached ||
                    eventually_altitude_reached.offset(by: 1).defaults(to: false)
output phi @(takeoff && altitude_reached):= !takeoff || eventually_altitude_reached
```

Here, the output stream `eventually_altitude_reached` recursively checks if the altitude is currently reached or will be reached in the future by using a future offset. The output stream `phi` then verifies the LTL property by ensuring that at every point in time, either no takeoff occurs, or altitude will eventually be reached. △

For these use cases, it is essential that the future offset operator integrates seamlessly into the existing RTLola framework. To achieve this, we extend the semantics of RTLola to incorporate future offsets, analyze their implications for front-end analysis, and ensure that specifications remain well-defined. In the following, we refer to the baseline language introduced in Sect. 3.2 as *RTLola* and the extended variant with future offsets as *RTLola+*. The subsequent sections present these extensions and their analyses in detail.

→ Sec. 3.2, p. 13

4.2. Semantics

In order to support future offsets in *RTLola+*, we must adapt the relational semantics of *RTLola* as defined in Sect. 3.2.1. Afterwards, we extend the inference rules for expressions and introduce the necessary functions.

→ Sec. 3.2.1, p. 17

For now, let us recall the semantics of *RTLola*, which are based on an evaluation model. *RTLola* uses a discrete time model, in which time progresses in discrete steps. Each stream in a specification is associated with a sequence of values indexed by these time steps. The semantics of *RTLola* is defined in terms of how these streams are evaluated at each time step depending on their definitions and dependencies.
In the baseline language, the set of all discrete time steps, denoted $\mathtt{Time}$, is not explicitly defined and is therefore assumed to be the set $\mathbb{N}$. This assumption suffices for RTLola, since no upper bound on time steps is required. However, to support future offsets, this assumption must be revised. For instance, consider the following code:

```
ouput x := x.offset(by: 1).defaults(to: 0)
```

Here, the output stream `x` references its own value at the next time step. Without an upper bound on the set of time steps, this definition would result in an infinite regress, as there is always another future step to consider. To prevent this issue, we redefine the set of discrete time steps $\mathtt{Time}^+$ as follows:

Definition 4.1 ($\mathtt{Time}^+$)

The set of all discrete time steps $\mathtt{Time}^+$ is defined as the set

$$\{t \in \mathbb{N} \mid 0 \leqslant t \leqslant T_{max}\}$$

where $T_{max} \in \mathbb{N}$ denotes the maximum time step.

This definition guarantees that the number of time steps is finite, ensuring that the evaluation of future offsets will eventually fall back to their default value. All other features of *RTLola* remain unaffected by this modification. Consequently, the evaluation model and semantics of *RTLola* can be reused for the extension *RTLola+*. The only difference is that, instead of assuming an unbounded set of discrete time steps, *RTLola+* employs the bounded set $\mathtt{Time}^+$.

The effect of this modification will be more visible when considering the inference rules, which are defined in Sect. A.1. In the baseline language *RTLola*, a specification may depend only on current or past values. Consequently, the inference rules for evaluating offset expressions capture only past offsets. Here, the inference rules for offsets specify that an `Offset` expression evaluates to the value of a stream at a given offset in the past. In this case, the `Default` expression evaluates to the evaluated value of the `Offset` expression. If insufficient past values are available, the expression defaults to $\bot$. This means that the `Default` expression must evaluate to the provided default value. These rules are formally given by:
EVAL-EXPR-OFF

$$\frac{\mathit{prefix} = \mathtt{Prefix}(\omega, \mathit{sid}, t) \quad |\mathit{prefix}| > \mathit{off} \quad \mathit{prefix}[|\mathit{prefix}| - \mathit{off}] = (t', v)}{\mathtt{Offset}(\mathit{sid}, \mathit{off}) \Downarrow_{\omega}^{t} v}$$

EVAL-EXPR-OFF-DFT

$$\frac{prefix = \texttt{Prefix}(\omega, sid, t) \quad |prefix| \leqslant off}{\texttt{Offset}(sid, off) \Downarrow^t_\omega \bot}$$

EVAL-EXPR-NO-DFT

$$\frac{expr \Downarrow^t_\omega v \quad v \in \mathbb{V}}{\texttt{Default}(expr, dft) \Downarrow^t_\omega v}$$

EVAL-EXPR-DFT

$$\frac{expr \Downarrow^t_\omega \bot \quad dft \Downarrow^t_\omega v}{\texttt{Default}(expr, dft) \Downarrow^t_\omega v}$$

To access past values, *RTLola* employs the recursive `Prefix` function, given by:

$$\texttt{Prefix} : \mathbb{W} \times \mathrm{ID}^* \times \mathrm{Time} \to (\mathrm{Time}, \mathbb{V})^*$$

$$\texttt{Prefix}(\omega, sid, t) := \begin{cases} (t, v) & \text{if } t = 0 \wedge \omega(sid)(t) = v \wedge v \in \mathbb{V} \\ \epsilon & \text{if } t = 0 \wedge \omega(sid)(t) = \bot \\ \texttt{Prefix}(\omega, sid, t-1) \cdot (t, v) & \text{if } \omega(sid)(t) = v \wedge v \in \mathbb{V} \\ \texttt{Prefix}(\omega, sid, t-1) & \text{if } \omega(sid)(t) = \bot \end{cases}$$

This function retrieves all time stamps and values of a stream up to a given time step t. For instance, consider Fig. 4.1 with a stream $StreamA$ with corresponding values $v_1 \ldots v_9$ up to $T_{max} = 9$. The behavior of the `Prefix` function at time step $t = 4$ is illustrated in the figure. The function returns all available values from time step 0 to 4, excluding any time steps where no value is present (e.g., time step 2). These rules and functions are sufficient for the baseline language, as defined in Sect. A.1.1 and Sect. A.2.1.
In contrast, a specification in the extended language *RTLola+* may also depend on future values. To support this, we introduce an additional recursive function that retrieves all time stamps and values of a stream from a given time step t up to the maximum time step T_{max}. This function is given by:

$$\texttt{Suffix} : \mathbb{W} \times \mathrm{ID}^* \times \mathrm{Time} \to (\mathrm{Time}, \mathbb{V})^*$$

$$\texttt{Suffix}(\omega, sid, t) := \begin{cases} (t, v) & \text{if } t = T_{max} \wedge \omega(sid)(t) = v \wedge v \in \mathbb{V} \\ \epsilon & \text{if } t = T_{max} \wedge \omega(sid)(t) = \bot \\ (t, v) \cdot \texttt{Suffix}(\omega, sid, t+1) & \text{if } \omega(sid)(t) = v \wedge v \in \mathbb{V} \\ \texttt{Suffix}(\omega, sid, t+1) & \text{if } \omega(sid)(t) = \bot \end{cases}$$

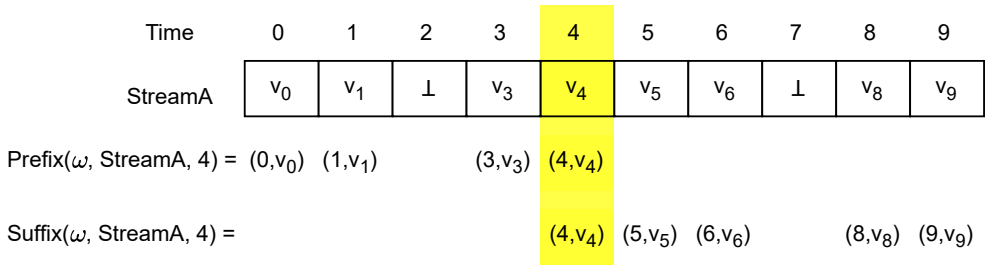


Figure 4.1.: Illustration of the `Prefix` and `Suffix` functions

The function returns a concatenation of tuples, each consisting of a time step and its corresponding value. For example, in Fig. 4.1, the behavior of the `Suffix` function at time step $t = 4$ is illustrated. The function returns all available values from time step 4 to 9, again excluding any time steps where no value is present (e.g., time step 7).

Based on the preceding function, we can now formalize the inference rules for evaluating offset expressions in *RTLola+*. To this end, it is necessary to distinguish between past and future offset expressions. The inference rules for offset expressions in the baseline language *RTLola* can be directly adopted for *RTLola+* and applied to past offsets. To emphasize this distinction, we rename the original `Offset` construct to `PastOffset`.
The next step is to introduce the inference rules for FutureOffset expressions. Analogous to the case of `PastOffset`, there are two inference rules for the evaluation of `FutureOffset` expressions, which are presented below.
`EVAL-EXPR-FUTURE-OFF:`

$$\frac{\mathit{suffix} = \texttt{Suffix}(\omega, \mathit{sid}, t) \quad |\mathit{suffix}| > \mathit{off} \quad \mathit{suffix}[\mathit{off}] = (t', v)}{\texttt{FutureOffset}(\mathit{sid}, \mathit{off}) \Downarrow_{\omega}^{t} v}$$

`EVAL-EXPR-FUTURE-OFF-DFT:`

$$\frac{\mathit{suffix} = \texttt{Suffix}(\omega, \mathit{sid}, t) \quad |\mathit{suffix}| \leqslant \mathit{off}}{\texttt{FutureOffset}(\mathit{sid}, \mathit{off}) \Downarrow_{\omega}^{t} \bot}$$

Instead of employing the `Prefix` function, we now use the `Suffix` function to obtain the set of all future values. Once this set is determined, a case distinction is required. If the number of available suffix elements exceeds the specified offset, the corresponding future value can be accessed by indexing into the set at the given offset. In this case, the first inference rule evaluates the expression to the retrieved value.
If, however, the specified offset exceeds the length of the suffix set, the last time step has already been reached and the desired value cannot be accessed. In this case, the second inference rule evaluates the expression to $\bot$, so that the default value must be taken.
For example, consider $\mathit{StreamA}$ and its corresponding `Suffix` function at time step $t = 4$, as illustrated in Fig. 4.1. The suffix set has length 5 in this case. Therefore, for a

`FutureOffset` expression with offset 4, the first rule applies and the evaluation yields v_9. For an offset of 5 or greater, the second rule applies and the evaluation yields $\bot$.

With the introduction of future offsets, the semantics of *RTLola* have been extended to form *RTLola+*. The modified inference rules for *RTLola+* are listed in Sect. A.1.2, together with the corresponding functions in Sect. A.2.2. Crucially, *RTLola*'s well-definedness property must also hold in this extended setting. By bounding the set of discrete time steps and adapting the inference rules accordingly, we ensure that specifications in *RTLola+* remain consistent and free of undefined behavior. The following chapter establishes how this well-definedness property is systematically ensured for *RTLola+*.

4.3. Dependency Analysis

The dependency analysis inspects the dependency graph of a specification, as defined in Def. 3.10. Its purpose is to guarantee a unique evaluation model and thereby satisfy the well-definedness criterion for *RTLola+*. To maintain analyzability, we structure the theoretical analysis into several semantic dimensions. This categorization allows us to isolate and address different semantic aspects systematically. In the subsequent sections, we consider the following semantic dimensions:

→ Def. 3.10, p. 18

- **Cycles:** Cycles in the dependency graph are classified as positive, negative, or zero. A positive cycle has an accumulated weight strictly greater than zero, a negative cycle has a weight less than zero and a zero cycle has a weight of exactly zero. Additionally, cycles may be asynchronous when combined with asynchronous operators, or a specification may contain no cycles at all.

- **Pacing:** Specifications may follow either event-based or periodic pacing. We further distinguish between synchronous pacing (all streams share the same pacing) and asynchronous pacing (streams operate at different rates).

- **Features:** Semantic filters may or may not be present in a specification.

A *RTLola+* specification can consist of any combination of these semantic categories. Our goal is to determine which combinations yield well-defined specifications. In particular, we want to impose specific conditions on the dependency graph based on these semantic dimensions. Therefore, we can define well-formed *RTLola+* specifications based on these conditions as an extension of the well-formedness properties of *RTLola*, as introduced in Def. 3.11. This property guarantees that all cycles in the dependency graph are either resolvable within the bounded time or excluded due to semantic ambiguity and therefore ensures well-definedness. The subsequent sections detail this analysis, outlining the admissibility conditions and providing representative examples to illustrate accepted and rejected patterns.

→ Def. 3.11, p. 20

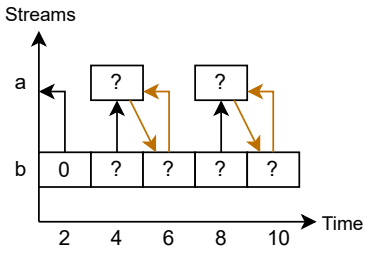
4.3.1. Analyzing Asynchronous Cycles

First, we will consider specifications with asynchronous pacings. This feature is the main difference compared to the origin language *Lola*, where future offsets exists. This is why, its well-formedness property must be re-evaluated in this context. In *RTLola+*, streams may operate at different rates, which can complicate the evaluation of future offsets. Therefore, the well-formedness property is insufficient for cycles in the dependency graph. We begin with examples based on static periodic pacings, because the evaluation behavior is more predictable due to the fixed rates.

Example 4.3.1. Consider the following specification:

```
output a @4s := b.offset(by: 1).defaults(to: 0)
output b @2s := a.hold(or: 0)
```

According to the well-formedness property of *RTLola*, this specification is well-formed, as it contains a positive cycle with accumulated weight of $+1$. However, consider the following trace:



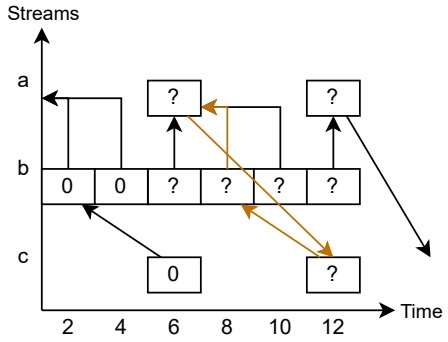
At each time step, both streams evaluate according to their respective periodic pacing. However, due to the asynchronous pacing, stream `b` evaluates twice as often as stream `a`. Therefore, at time step $t = 6s$ the `Hold` operator in stream `b` accesses the last seen value from stream `a` at time step $t = 4s$. Here, at time step $t = 4s$ stream `a` accesses stream `b` at time step $t = 6s$ with a future offset. This leads to an evaluation deadlock, as both streams want to resolve each other. The conclusion is that there does not exist a unique evaluation model for this specification, s.t. it is not well-defined. △

Based on this obsersavtion, we know that the well-formedness property from *RTLola* is not sufficient to guarantee well-definedness in *RTLola+*. Therefore, we need to refine the conditions of the well-formedness property in *RTLola+* to also cover asynchronous cycles with future offsets. Moreover, there are also negative asynchronous cycles that lead to similar issues.

Example 4.3.2. Consider the following specification:

```
output a @6s := c.offset(by: 1).defaults(to: 0)
output b @2s := a.hold(or: 0)
output c @6s := b.offset(by: -2).defaults(to: 0)
```

Here, we have a negative cycle with total weight of -1. However, consider the following trace:



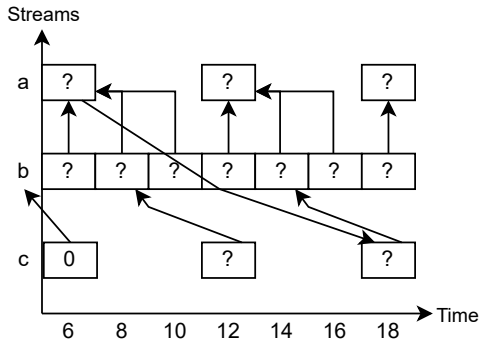
We can observer here, that stream `b` evaluates three times as often as stream stream `a` and `c`. At time step $t = 6s$, stream `a` accesses stream `c` at time step $t = 12s$ with a future offset. Here, stream `c` accesses stream `b` at time step $t = 8s$, while stream `b` accesses stream `a` at time step $t = 6s$. This leads to an evaluation deadlock, as all streams want to resolve each other. The conclusion is that there does not exist a unique evaluation model for this specification, s.t. it is not well-defined. △

These examples illustrate that both positive and negative asynchronous cycles also can lead to ill-defined specifications in *RTLola+*. In other cases however, where a specification should be rejected, the evaluation model is unique and well-defined.

Example 4.3.3. Consider the following specification:

```
output a @6s := c.offset(by: 2).defaults(to: 0)
output b @2s := a.hold(or: 0)
output c @6s := b.offset(by: -2).defaults(to: 0)
```

Here, we have a in fact a cycle with total weight of 0. Due to the well-formedness property, this specification should be rejected. However, consider the following trace:



Unlike the previous examples, this specification does not lead to an evaluation deadlock. We can observe that at each time step, all streams can successfully resolve their dependencies. Therefore, there exists a unique evaluation model for this specification, s.t. it is well-defined. △

This example illustrates that even zero cycles can be well-defined in this setting. However, this behavior only occurs in very specific cases, where the offsets align perfectly with the pacing rates. For this reason, we need to be cautious when designing specifications with asynchronous cycles. In general, we need to impose restrictions on asynchronous cycles in *RTLola+* to ensure well-formedness.

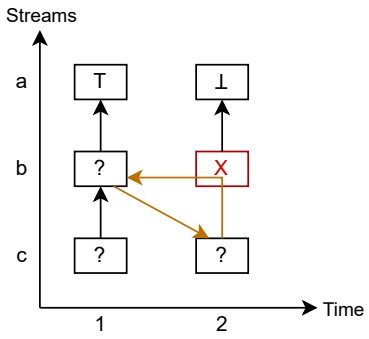
The previous examples are based on periodic pacings. However, these issues also arise with event-based pacings. The main difference is that we do not know the exact evaluation time steps in advance. Here, the same reasoning applies, as there are input sequences that produce the same evaluation deadlocks as shown in the examples above. Since the well-definedness property is defined for every possible input sequence, these specifications are also not well-defined in the well-formed fragment of *RTLola+*.

Beside these pacing types, we consider specifications with semantic filters. Here, we get a dynamic semantic filter in addition to the static pacing type. The static pacing type is determined by the declared pacing of the stream, while the dynamic semantic filter may enable or disable the evaluation of a stream at runtime. Thus, the prescence of the semantic filter produces another layer of asynchronounity.

Example 4.3.4. Consider the following specification:

```
input a : Bool
output b eval @a when a with c.offset(by: 1).defaults(to: false)
output c @a := b.hold(or: false)
```

Here, we have a positive cycle with total weight of $+1$. Both output streams evaulate, if stream `a` receives a value. Moreover, the stream `b` only evaluates when stream `a` is true and the semantic filter evaluates to true. However, consider the following trace:



We can observe here, that stream `b` evauluates at time step $t = 1$, since semantic filter evaluates to true, s.t. stream `c` will be accessed at time step $t = 2$ with a future offset

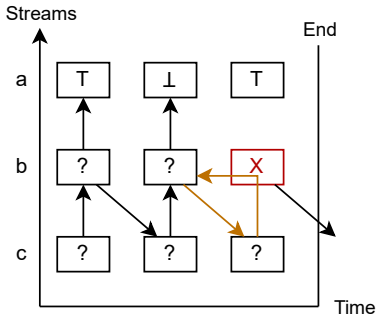
operator. At time step $t = 2$, stream `b` will not evaluate, since the semantic filter evaluates to false. Therefore, stream `c` will access the last seen value from stream `b` at time step $t = 1$. This leads to an evaluation deadlock, as both streams want to resolve each other. The conclusion is that there does not exist a unique evaluation model for this specification, s.t. it is not well-defined. △

This example illustrates that asynchronous cycles with semantic filters can also lead to ill-defined specifications in *RTLola+*. Even though there is a cycle with a static synchronous pacing, the semantic filter introduces an additional layer of asynchronounity. Therefore, the same reasoning as above apply, as there are input sequences that produce the same evaluation deadlocks as shown in the examples. Since the well-definedness property is defined for every possible input sequence, these specifications are also not well-defined. Therefore, we need to impose restrictions on all asynchronous cycles in *RTLola+* to ensure well-formedness.
The same reasoning also applies to specifications containing a future offset in a filter condition.

Example 4.3.5. Consider the follwoing specification:

```
input a: Bool
output b eval @a when c.offset(by: 1).defaults(to: false) with a
output c @a := b.hold(or: false)
```

Here, we have a cycle that contains a filter edge and an offset edge. However, consider the following trace:



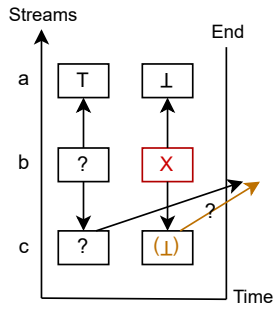
In this case, the evaluation deadlock arises at the end of the trace T_{max}. Here, the semantic filter of stream `b` evaluates to its default value, which is false. Therefore stream `b` will not evaluate at this time step. Thus, stream `c` will access the last seen value from stream `b`. However, the last seen value from stream `b` accesses stream `c` at this time step with a future offset. This leads to an evaluation deadlock, as both streams want to resolve each other. The conclusion is that there does not exist a unique evaluation model for this specification, s.t. it is not well-defined. △

Other cases, where we have cycles with filter edges and offset edges, are already disallowed by the well-formedness property of *RTLola*.

Example 4.3.6. Consider the following specification:

```
input a: Bool
output b eval @a when c with a
output c @a := b.offset(by: 1).defaults(to: false)
```

Here, we have a cycle that contains a filter edge and an offset edge. According to the well-formedness property of *RTLola*, this specification is not well-formed, as the offset edge is outside the semantic filter. Consider the following trace:



Here, the uncertainty arises at time step T_{max}. At this time step, the stream `c` evaluates to its default value, which is false. Therefore stream `b` will not evaluate at this time step. This forms an contradiction, since stream `c` tries to access stream `b` at this time step with a future offset. However, there is no reference point to perform an discrete future offset from. Therefore, this specification is rejected in the well-formed fragment of *RTLola+*. △

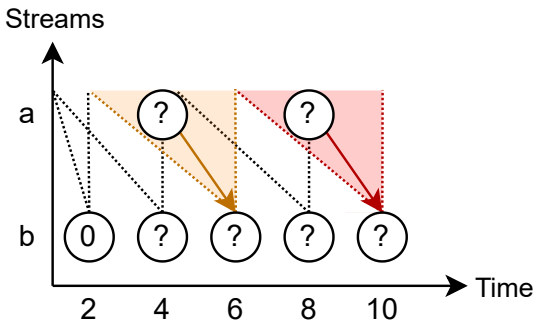
Based on these observations, we need to impose restrictions on all asynchronous cycles that contain future offsets in *RTLola+* to ensure well-formedness.

To this end, we focused on asynchronous cycles with the `Hold` operator. However, similar issues also arise with the other asynchronous operator, namely aggregations over sliding windows. In this context, future offsets lead to evaluation deadlocks in a similar manner.

Example 4.3.7. Consider the following specification:

```
output a @4s := b.offset(by: 1).defaults(to: 0)
output b @2s := a.aggregate(over: 4s, using: sum)
```

Here, we have a positive cycle. To get the problematic behavior, we consider the following trace:



In this case, stream `b` aggregates over the last 4s at time step $t = 6s$, including the the value from stream `a` at time step $t = 4s$. However, at time step $t = 4s$, stream `a` accesses stream `b` at time step $t = 6s$ with a future offset. This leads to an evaluation deadlock, as both streams want to resolve each other. The conclusion is that there does not exist a unique evaluation model for this specification, s.t. it is not well-defined. △

This example illustrates that asynchronous cycles with aggregations and future offsets can also lead to ill-defined specifications in *RTLola+*. The same reasoning also applies to synchronous pacings, since the aggregation operator evaluates over a time window in the past. Therefore, if the evaluation of an future offset occurs within the aggregation window, the same evaluation deadlock arises as shown in the example above. Again, these cases only occure, if the pacings, offsets and aggregation windows align perfectly. For instance, in Example 4.3.7, if the aggregation window was 2s instead of 4s, the specification would be well-defined, since the future offset would not fall within the aggregation window. Also, if the offset was 2 instead of 1, the specification would be well-defined, since the future offset would access a time step outside of the aggregation window. Therefore, we need to consider cycles with aggregations and future offsets in the same manner as cycles with the `Hold` operator, but without the pacing restrictions.

Remark 4.3.1. *The previous examples showed us that future offsets in combination with asynchronous operators can lead to ill-defined specifications. However, there are predictable partterns that allow such specifications to be well-defined (e.g., Example 4.3.3). The detection of these patterns is cheap for small specifications, but becomes increasingly complex for more complex specifications. Therefore, we decided to exclude all asynchronous cycles with future offsets in* RTLola+ *to ensure well-definedness. This restriction simplifies the analysis and guarantees that specifications in the well-formed fragment are free from evaluation deadlocks caused by future offsets. The analysis of these predictable patterns is left for future work.*

4.3.2. Analyzing Synchronous Cycles

Next, we consider specifications with a synchronous pacing. In this case all streams have either equal periodic or event-based pacing, s.t. the stream evaluations occur

simultaneously. Here, the well-formedness property directly applies, as the presence of future offsets does not introduce additional complications. With synchronous pacings, the evaluations appear to be just as in *Lola*. Also, the `Hold` operator does not introduce any asynchronous behavior in this setting and can be treated as an synchronous access.

Example 4.3.8. Consider the following specification:

```
input x: Int
output a @x := b.offset(by: 1).defaults(to: 0)
output b @x := a.hold(or: 0)
```

This specification is well-formed, as it contains a positive cycle with accumulated weight of $+1$. Both output streams `a` and `b` are evaluating at the same event-based pacing, triggered by the input stream `x`. Therefore, the `Hold` operator in stream `b` will always access the last evaluated value of stream `a`, which is always the value from the current time step. The equivalent specification would be:

```
input x: Int
output a @x := b.offset(by: 1).defaults(to: 0)
output b @x := a
```

Thus, the cycle can be resolved at time step T_{max}, where stream `a` accesses its default value. Afterwards the cycle can be resolved backwards. △

This example illustrates that specifications with synchronous pacings can be analyzed using the well-formedness property from *RTLola*. Besides positive cycles, negative cycles in combination with future offsets are also allowed in this setting and do not need further restrictions. As mentioned before, *Lola* already layed down the theoretical foundation for future offsets in a synchronous setting [11]. However, certain cycle types are disallowed in this setting due to the well-formedness conditions.

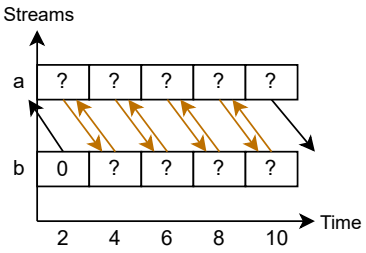
Example 4.3.9. Consider the following specification:

```
input x : Bool
output a @x := b.offset(by: 1).defaults(to: 0)
output b @x := a.offset(by: -1).defaults(to: 0)
```

This forms a zero cycle with accumulated weight of 0. Such cycles are disallowed, since they violate the first condition of the well-formedness property of *RTLola+*.
The same observation holds for periodic pacings:

```
output a @2s := b.offset(by: 1).defaults(to: 0)
output b @2s := a.offset(by: -1).defaults(to: 0)
```

This behavoir can be illustrated with the following trace:



Here, at each time step both streams evaluate according to their periodic pacing. However, due to the zero cycle, stream `a` accesses stream `b` at the next time step, while stream `b` accesses stream `a` at the previous time step. This leads to an evaluation deadlock, as both streams want to resolve each other. The conclusion is that there does not exist a unique evaluation model for this specification, s.t. it is not well-defined. △

This example illustrates that zero cycles with future offsets are disallowed in *RTLola+*, as they violate the well-formedness property. Furthermore, the synchronous behavior does not only apply to equal event-based pacings, but also to periodic pacings. As long as the periods are equal, the same reasoning applies, since the streams evaluate at the same time steps, which implies a synchronous behavior.

4.3.3. The Well-Formedness Property

Based on these observations, we exclude asynchronous cycles with future offsets for *RTLola+*. Therefore, we refine the well-formedness property from *RTLola*, as introduced in Def. 3.11, and adapt it to the extended setting of *RTLola+*.

→ Def. 3.11, p. 20

Definition 4.2 (*RTLola+* Well-Formed)

Def. *RTLola+* Well-Formed

Given a *RTLola+* specification Φ and its corresponding dependency graph $D_\Phi = \langle V, E \rangle$. The RTLola specification Φ is called *well-formed*, if the dependency graph D_Φ does not contain any cycles $C = \{(s_1, (t_1, l_1), s_2), \ldots, (s_n, (t_n, l_n), s_1)\} \subseteq E$:

- with the accumulated edge weight of zero.
- that contain an $\mathtt{Filter}$-edge and an $\mathtt{Offset}$-edge:

$$\forall C : \exists i : l_i = (\mathtt{Offset}, \_) \wedge \forall j : t_j \neq \mathtt{Filter}$$

- that contain an positive $\mathtt{Offset}$-edge and an $\mathtt{Aggregate}$-edge:

$$\forall C : \exists i : l_i = (\mathtt{Offset}, \mathtt{off}) \wedge \mathtt{off} > 0 \wedge \forall j : l_j \neq (\mathtt{Aggregate}, \_, \_)$$

- that contain an positive $\mathtt{Offset}$-edge and an $\mathtt{Hold}$-edge with asynchronous pacing:

$$\forall C : \exists i : l_i = (\mathtt{Offset}, \mathtt{off}) \wedge \mathtt{off} > 0 \wedge \forall j : l_j \neq \mathtt{Hold} \wedge \mathtt{Pace}(s_i) \neq \mathtt{Pace}(s_j)$$

This holds under the assumption, that for each edge $(s, (t, l), s') \in E$, both streams have either periodic or event-based pacings. The function `Pace(s)` returns the static pacing type as well as the dynamic semantic filter of a stream s.

With this refinement, the well-formedness property of *RTLola+* must guarantee a unique evaluation model for its specifications.

In order to prove the uniqueness, we consider a trace as a directed graph, which we will call the *evaluation graph* EG. This structure is inspired by a similar construction in *Lola* [11]. Here, each node in this graph can be identified by a tuple (sid, ts), where sid is the stream identifier and ts is the time step. A directed edge exists from node (sid_1, ts_1) to node (sid_2, ts_2) if the evaluation of stream sid_1 at time step ts_1 depends on the value of stream sid_2 at time step ts_2. The construction of this evaluation graph is based on the dependency graph of the specification and a specific trace.

Def. Evaluation Graph

The following poofs of unique evaluation models for well-formed *RTLola+* specifications are inspired by similar proofs for *Lola* [11]. However, we need to adapt these proofs to the extended setting of *RTLola+* and the well-formedness property defined above. The first lemma establishes a connection between cycles in the evaluation graph and violations of the well-formedness property in the dependency graph.

Lemma 1. *Given a* RTLola+ *specification* Φ *and its corresponding dependency graph* D_Φ*. Let* EG_Φ *be the evaluation graph of* Φ *with trace length* T_{max}*. If there exists a cycle in* EG_Φ*, then there exists a corresponding cycle in* D_Φ *that violates the well-formedness property from Def. 4.2.*

Proof. Assume there exists a cycle in the evaluation graph EG_Φ. This cycle consists of a sequence of nodes $(sid_1, ts_1), \ldots, (sid_n, ts_n), (sid_1, ts_1)$. Each directed edge $(sid_i, ts_i) \to (sid_{i+1}, ts_{i+1})$ in this cycle corresponds to an edge $(sid_i, (t_i, l_i), sid_{i+1})$ in the dependency graph D_Φ. By summing the timestep differences around the cycle in EG_Φ yields $\sum_{i=1}^{n}(ts_{i+1} - ts_i) = 0$. This implies that the corresponding cycle in D_Φ contains edges that violate some well-formedness conditions outlined in Def. 4.2. Particularly, we must proof that the cycle in D_Φ must fall into one of the following categories: Suppose that the corresponding cycle in D_Φ does not violate the well-formedness property. Then, the corresponding evaluation graph EG_Φ will not contain a cycle. Therefore, we analyze all possible cycle types in D_Φ based on the semantic dimensions and show that they lead to a contradiction with our assumption.

- **Asynchronous Cycles:** In this setting, the cycle in D_Φ must contain at least one of both asynchronous operators, i.e., `Hold` or `Aggregate`.
  For the first case, the `Hold` operator introduces an asynchronous behavior, which leads to an evaluation deadlock in combination with future offsets. This applies to all asynchronous cycle types that contain future offsets and the `Hold` operator, as shown in Examples 4.3.1, 4.3.2, 4.3.4 and 4.3.5. Therefore, the well-formedness property is violated.

For the second case, the `Aggregate` operator introduces an asynchronous behavior, which leads to an evaluation deadlock in combination with future offsets, as shown in Example 4.3.7. Thus, the well-formedness property is violated.

- **Synchronous Cycles:** The synchronous setting behaves similar to *Lola*, as the synchronous pacing eliminates the asynchronous behavior of the `Hold` operator. This means, that the `Hold` operator can be treated as an synchronous access as seen in Example 4.3.8. Therefore, both positive and negative cycles are allowed in this setting. However, zero cycles lead to evaluation deadlocks, as shown in Example 4.3.9. Thus, the well-formedness property is violated.
  The `Aggregate` operator may still lead to ill-defined specifications in combination with future offsets. Therefore, cycles that contain both an positive offset edge and an aggregate edge can lead to evaluation deadlocks, as shown in Example 4.3.7. Thus, the well-formedness property is violated.

- **Semantic Filter Cycles:** These cycles are disallowed by the well-formedness property for both asynchronous and synchronous setting. This is because they lead to ill-defined specifications, as shown in Example 4.3.6.

Therefore, the existence of a cycle in $\mathtt{EG}_\Phi$ implies the existence of a cycle in $\mathtt{D}_\Phi$ that violates the well-formedness property. □

This lemma establishes a direct relationship between cycles in the evaluation graph and violations of the well-formedness property in the dependency graph. Note, that in the examples above, we illustrated cycles in the trace that lead to ill-defined specifications. According to this lemma, these cycles correspond to cycles in the dependency graph that violate the well-formedness property from Def. 4.2. Next, we show that if the evaluation graph is acyclic, then an unique evaluation model exists.

Lemma 2. *Given a* RTLola+ *specification* Φ *and its corresponding dependency graph* $\mathtt{D}_\Phi$*. Let* $\mathtt{EG}_\Phi$ *be the evaluation graph of* Φ *with trace length* $\mathtt{T}_{max}$*. If* $\mathtt{EG}_\Phi$ *does not contain any cyles, then there exists a unique evaluation model for* Φ *for every possible input trace of length* $\mathtt{T}_{max}$*.*

Proof. If the evaluation graph $\mathtt{EG}_\Phi$ does not contain any cycles, it implies that there are no circular dependencies among the stream evaluations for the given trace length $\mathtt{T}_{max}$. This acyclic structure allows us to perform a topological sort on the nodes of $\mathtt{EG}_\Phi$, which provides a linear ordering of stream evaluations. By following this ordering, we can evaluate each stream at each time step in a manner that respects all dependencies. Since there are no cycles, each stream value at a given time step can be computed based on previously computed values or remains unresolved. Consequently, this guarantees the existence of a unique evaluation model for the specification Φ for every possible input trace of length $\mathtt{T}_{max}$. □

This lemma confirms that an acyclic evaluation graph guarantees a unique evaluation model for the specification. To apply Lem. 2, we must ensure that the evaluation graph

is acyclic for all possible input traces of length T_{max}. This is blocked by cycles in the evaluation graph, which we addressed in Lem. 1. For instance, consider the examples provided earlier. Each well-formedness property targets a specific type of cycle in the dependency graph that could lead to ill-defined specifications. By ensuring that the dependency graph does not contain such cycles, we can guarantee that the evaluation graph remains acyclic for all possible input traces.

Combining both lemmas, we can now prove that well-formed specifications in *RTLola+* guarantee a well-defined specification.

Theorem 3. *Each* RTLola+ *specification* Φ *that is well-formed according to Def. 4.2 guarantees a well-defined specification according to Def. 3.9* → Def. 3.9, p. 18

Proof. Given a *RTLola+* specification Φ that is well-formed, we need to show that it guarantees a well-defined specification. By contraposition, we assume that Φ does not guarantee a well-defined specification. This implies that there exists at least one input trace of length T_{max} for which there is no unique evaluation model. According to Lem. 2, this lack of a unique evaluation model indicates that the evaluation graph EG_Φ must contain a cycle for that specific trace. However, by Lem. 1, the presence of a cycle in EG_Φ implies the existence of a corresponding cycle in the dependency graph D_Φ that violates the well-formedness property outlined in Def. 4.2. This contradicts our initial assumption that Φ is well-formed. Therefore, we conclude that if Φ is well-formed, it must guarantee a well-defined specification. □

Note, that the converse of this theorem does not hold, i.e., Example 3.1.4. There exist specifications that are well-defined, but not well-formed according to Def. 4.2. However, these specifications are rare and difficult to identify. Therefore, we accept this limitation to ensure a simpler and more efficient analysis for the well-formed fragment of *RTLola+*. → Ex. 3.1.4, p. 11

Remark 4.3.2. *These proofs are inspired by similar proofs for* Lola *[11]. However, we had to adapt them to account for the additional complexities introduced by future offsets and asynchronous pacings in* RTLola+. *The core ideas remain similar, but the specific conditions and implications had to be carefully reconsidered to ensure correctness in this extended context.*

4.4. Order Analysis

The Order analysis determines a valid evaluation sequence for streams at each time step, ensuring that all dependencies are resolved before a stream is computed. However, the order analysis from *RTLola* is no longer sufficient, as future offsets introduce delays in the evaluation of certain streams. Therefore, we will inspect the order analysis from *Lola* [11], which already addresses future offsets for synchronous streams, and adapt it to our asynchronous setting in *RTLola+*. Here, we consider the Rust implementation of *Lola* [18] as a basis for the order analysis.
First, we assume that the specification is well-formed according to Def. 4.2, s.t. a unique

evaluation model exists. Next, we notice that the presence of future offsets may delay the resolution of certain stream values. Therefore, we introduce the notion of *delay* to capture the maximum number of evaluation steps a stream value remains unresolved due to future offsets. Before that, we need to define the *Delay Weight* for edges in the dependency graph in order to compute the delay for each stream.

Definition 4.3 (Delay Weight)

Given a *RTLola+* specification Φ and its corresponding dependency graph D_Φ. We define the *Delay Weight* for each edge $(s, (t, l), s') \in E$ as a function $\omega : (ID^*, ID^*) \to \mathbb{Z}$:

$$\omega(s, s') = \begin{cases} n & \text{if } (s, (\_, (\mathtt{Offset}, n)), s') \in E \\ 0 & \text{otherwise} \end{cases}$$

This function assigns a weight of n to edges representing offsets and a weight of 0 to all other edges.

Here, we only consider edges with offset operators, as they are the only ones that can influence delays in the evaluation. Based on this, we can now define the delay for each stream in the dependency graph, which is inspired by the delay notion from the Rust implementation of *Lola* [18].

Definition 4.4 (Delay)

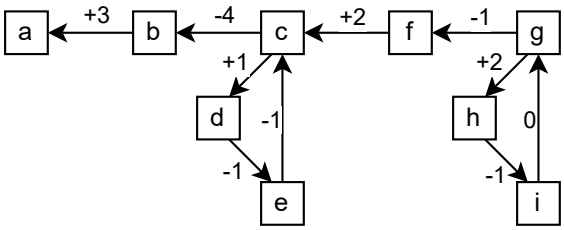
Given a *RTLola+* specification Φ and its corresponding dependency graph D_Φ. We define the *Delay* $\triangle(s)$ for each stream $s \in V$ as the maximum number of evaluation steps a value of stream s will remain unresolved due to future offsets. Formally:

$$\triangle(s) = \max\left\{0, \max\left\{\omega(s, s') + \triangle(s') \mid (s, (\_, \_), s') \in E\right\}\right\}$$

Here, we compute the maximum weight of all possible paths starting from stream s.

Thus, we get $\triangle(s) = +\infty$, if there is a reachable cycle of positive total weight. Otherwise $\triangle(s)$ is the maximum finite accumulated offset, or 0 if all reachable path sums are less than or equal to zero.

Example 4.4.1. Consider the following dependency graph, where the edge weights are already translated with the function ω:



Now, we compute the delay of each stream:

σ	a	b	c	d	e	f	g	h	i
$\triangle(\sigma)$	0	3	1	0	0	3	∞	∞	∞

First, we observe that streams `g`, `h` and `i` have an infinite delay, since they are part of a positive cycle. Therefore, their values can only be resolved at time step T_{max}. Intuitively, this means that these streams will always access their own values at future time steps, leading to an infinite delay.
Next, we consider the negative cycle around the streams `c`, `d` and `e`. Here, we can observe that stream `c` has a delay of 1. This means, that the value of stream `c` will remain unresolved for 1 time step due to the future offset. Streams `d` and `e` have a delay of 0, since the total weights of all their paths lead to a value less than zero.
Finally, we consider streams `a`, `b` and `f`. Here, stream `b` has a delay of 3, since it accesses stream `a` with an offset of $+3$. Stream `f` also has a delay of 3, since it accesses stream `d` through stream `c` with an total offset of 3. Stream `a` has a delay of 0, since all its reachable paths lead to a value less than zero. $\triangle$

Intuitively, the delay captures the maximum number of time steps a stream value remains unresolved due to future offsets. With this information, we can now adapt the order analysis from *Lola* [18] to account for future offsets in *RTLola+*. Therefore, we need to modify the dependency graph by taking the delays into account. Here, we introduce *delay edges* to the dependency graph, which connect streams based on their delays. Formally, we define the *Delay Dependency Graph* $D_\Phi^\triangle$ as follows:

Definition 4.5 (Delay Dependency Graph [18])

Given a *RTLola+* specification Φ and its corresponding dependency graph $D_\Phi = \langle V, E \rangle$. We define the *Delay Dependency Graph* $D_\Phi^\triangle$ as follows:

Def. Delay Dependency Graph

$$D_\Phi^\triangle = \left\langle V, E^\triangle = \{(s, \triangle(s) - \omega(s, s') - \triangle(s'), s') \mid (s, \_, s') \in E\} \right\rangle$$

The resulting graph $D_\Phi^\triangle$ is a modified version of the dependency graph D_Φ that incorporates delay edges based on the computed delays of each stream.

In this definition, we modify the edges between streams s and s' with a weight of $\triangle(s) - \omega(s, s') - \triangle(s')$. This weight captures the difference in delays between the two streams, adjusted by the offset weight of the edge connecting them. With this modified dependency graph, we can now determine a valid evaluation order for the streams at each time step. This ensures that all dependencies are resolved before a stream is computed, taking into account any delays introduced by future offsets.

Definition 4.6 (*RTLola+* Evaluation Order)

Def. *RTLola+* Evaluation Order

Given a *RTLola+* specification Φ and its corresponding dependency graph D_Φ. We define the *Evaluation Order* $\prec \subseteq ID^* \times ID^*$ as the partial order on streams with the delay dependency graph $D_\Phi^\triangle$, which must satisfy the following conditions:

1. $\forall in \in ID^{\uparrow} : \forall out \in ID^{\downarrow} : in \prec out$
2. $\forall (o, (t, l), o') \in E : l \neq (Offset, \_) \implies o' \prec o$
3. $\forall (s, (t, l), s') \in E : (s, 0, s') \in E^\triangle \implies s' \prec s$
4. $\forall (s, (t_1, l_1), s') \in E, \forall (s', (t_2, l_2), s'') \in E :$

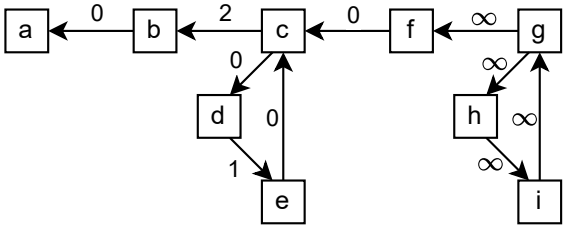
$$((t_1, l_1) = (Eval, (Offset, \_))) \wedge ((t_2, l_2) = (Filter, \_)) \implies s'' \prec s$$

Triggers are evaluated after all streams, as they are not accessed by any other streams.

This definition extends the order analysis from *Lola* [18] and combines it with the evaluation order from *RTLola* in Def. 3.12 to account for future offsets in *RTLola+*. The first condition ensures that input streams are evaluated before output streams. The second condition guarantees that if there is a direct dependency between two streams without an offset, the dependent stream is evaluated after the stream it depends on. The third condition guarantees that if there is a direct dependency between two streams with zero delay difference, the dependent stream is evaluated after the stream it depends on. The fourth condition addresses cases where a stream depends on another stream through an offset followed by a filter, ensuring that the evaluation order respects this indirect dependency. By following this evaluation order, we can ensure that all dependencies are resolved before a stream is computed, even in the presence of future offsets.

→ Def. 3.12, p. 21

Example 4.4.2. Consider the dependency graph from Example 4.4.1. Now, we compute the delay dependency graph:



Here, we can observe that the edges have been modified based on the delays of the streams. Note, that the edges between streams `g`, `h` and `i` have a weight of ∞, since they are part of a positive cycle. Therefore, these streams can only be evaluated at time

step T_{max} and are not part of the new condition for the evaluation order. However the second condtition still applies, since there are direct dependencies between these streams without offsets. Therefore, we obtain: $g \prec i$ For the other stream, we can determine a valid evaluation order based on the third condition:

$$a \prec b \quad d \prec c \quad c \prec e \quad c \prec f$$

This evaluation order ensures that all dependencies are resolved before a stream is computed, taking into account the delays introduced by future offsets. For instance, stream `b` is evaluated after stream `a`, since it depends on `a` with an offset of $+3$. Similarly, stream `c` is evaluated after stream `d`, as it depends on `d` with an offset of $+1$. △

This example illustrates how the order analysis can be adapted to account for future offsets in *RTLola+*. By modifying the dependency graph to include delay edges, we can determine a valid evaluation order that respects all dependencies, ensuring correct stream evaluations at each time step. Based on this evaluation order, we can determine the evaluation layers for streams at each time step. The evaluation layers do not require any modifications, as they are directly derived from the evaluation order. Therefore, we can directly apply the evaluation layer definition from Def. 3.13 of *RTLola* to *RTLola+*. → Def. 3.13, p. 22
For instance, in the previous example, we can determine the following evaluation layers:

$$\begin{aligned} \mathrm{Layer}_{\prec}(a) &= 0 \\ \mathrm{Layer}_{\prec}(b) = \mathrm{Layer}_{\prec}(d) = \mathrm{Layer}_{\prec}(g) = \mathrm{Layer}_{\prec}(h) &= 1 \\ \mathrm{Layer}_{\prec}(c) = \mathrm{Layer}_{\prec}(i) &= 2 \\ \mathrm{Layer}_{\prec}(e) = \mathrm{Layer}_{\prec}(f) &= 3 \end{aligned}$$

Remark 4.4.1. *Sliding windows are not explicitly considered in this order analysis. These kind of operations need to be treated separately, as they introduce additional dependencies based on the time window. This behavior and the general correctness of the order analysis must be further investigated in future work.*

4.5. Memory Analysis

The memory analysis estimates the memory requirements for streams in a specification. This analysis is crucial for ensuring that specifications can be monitored efficiently, particularly for online monitoring. The presence of future offsets introduces new considerations for memory usage, as streams may need to retain values for future evaluation. In the subsequent sections, we will first explore the efficiently monitorable fragement on the basis of *Lola* [11, 18] and then adapt the memory analysis to account for future offsets in *RTLola+*.

4.5.1. Efficiently Monitorable Fragement

The efficiently monitorable fragment must satisfy the well-formedness property from Def. 4.2 to guarantee a unique evaluation model. Therefore, all restrictions from the well-formedness property also apply to the efficiently monitorable fragment. In addition, the efficiently monitorable fragment imposes further restrictions to satisfy the efficiently monitorable property based on Def. 3.6 from *Lola* [11]. In order to define this property for *RTLola+*, we introduce the notion of *bounded* and *unbounded* memory usage:

→ Def. 3.6, p. 13

Definition 4.7 (Bounded and Unbounded Streams)
Given a well-formed *RTLola+* specification Φ. A stream s in Φ is called *bounded*, if the memory required to store its values does not grow with the length of the input trace. Otherwise, the stream s is called *unbounded*.

Based on this, we can now define the efficiently monitorable property for *RTLola+*.

Definition 4.8 (Efficiently Monitorable *RTLola+* Specifications [11])
A well-formed *RTLola+* specification Φ is called *efficiently monitorable*, if for every possible input trace, the memory usage for all streams is *bounded*. Otherwise, there exists at least one stream whose memory usage grows *unbounded* with the trace length.

This definition extends the efficiently monitorable property from *Lola* [11] in terms of bounded and unbounded memory usage. However, the presence of future offsets introduces new challenges for ensuring bounded memory usage. In order to analyze these challenges, we will first consider specifications with synchronous pacings. In this setting, the analysis from *Lola* [11] already provides a solid foundation, as it addresses future offsets in a synchronous context.

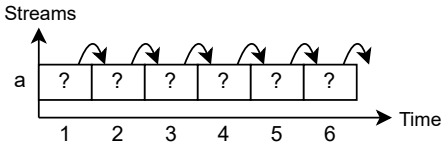
Theorem 4. *If the dependency graph of a* RTLola+ *specification has a positive cycle then the streams in this cycle are unbounded.*

The converse of this theorem does not hold [11]. However, this theorem establishes that specifications with positive cycles in their dependency graph are not efficiently monitorable. The presence of future offsets in these cycles can lead to situations where streams must retain values for future evaluation, resulting in unbounded memory usage. For instance, consider Example 4.3.8. Here, the specification contains a positive cycle with accumulated weight of $+1$. This means that at every evaluation time step, the vaule of stream `a` depends on the value of stream `b` at the next time step. This loop continues until the maximum time step T_{max} is reached, where stream `a` accesses its default value and the cycle can be resolved backwards. However, this implies that stream `a` must store all intermediate values until T_{max} is reached, leading to a memory usage that depends on the trace length. Therefore, this specification is not efficiently monitorable. The same observation holds also for positive cycles with periodic pacings.

Example 4.5.1. Consider the following specification:

```
output a @1s := a.offset(by: 1).defaults(to: 0)
```

Here, we have a positive cycle with accumulated weight of $+1$. According to Def. 4.2, this specification is well-formed. However, consider the following trace:



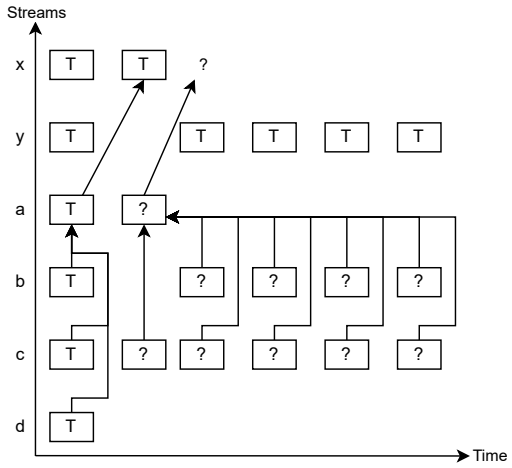
We can observe that stream `a` evaluates at each time step according to its periodic pacing of 1s. Therefore, at each time step, stream `a` accesses its own value at the next time step with a future offset. This loop continues until the maximum time step T_{max} is reached, where stream `a` accesses its default value. Consequently, stream `a` must store all unresolved values until then, leading to a unbounded memory usage. Therefore this specification is not efficiently monitorable. △

This example illustrates that specifications with positive cycles are not efficiently monitorable in *RTLola+*. The presence of future offsets in the cycle leads to a situation where streams must retain values for future evaluation, resulting in a trace length dependent memory usage. Therefore, we need to refine the efficiently monitorable property for *RTLola+* to exclude such cases. All other observations in this synchronous setting directly carry over from *Lola* [11] and do not require further restrictions. Thus, we can guarantee an bounded memory usage for specifications in the efficiently monitorable fragment of *RTLola+* with synchronous pacings.
Next, we will consider specifications with asynchronous pacings. Here, Thm. 4 still applies, but there are more restrictions to consider. In this case we do not only have to consider cycles in general, but also specifications with different pacing types and without cycles.

Example 4.5.2. Consider the following specification:

```
input x: Bool
input y: Bool
output a @x := x.offset(by: 1).defaults(to:false)
output b @y := a.hold(or: false)
output c @(x||y) := a.hold(or: false)
output d @(x&&y) := a.hold(or: false)
```

Here, we do not have any cycles in the dependency graph. Therefore it is well-formed according to Def. 4.2. However, consider the following trace:



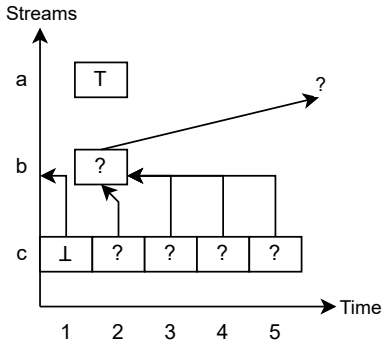
We can observe that stream `a` evaluates every time stream `x` delivers a value with the next value of stream `x`. Therefore, if stream `x` stops delivering values, the last evaluation of stream `a` stays unresolved until T_{max}. Consequently, streams `b` and `c` continuously access the last seen value from stream `a` at their respective event-based pacings. Since both streams could stay unresolved until T_{max} is reached, they must store all values until then, leading to an unbounded memory usage. Therefore this specification is not efficiently monitorable. However, stream `d` does not suffer from this issue, since it only evaluates when both streams `x` and `y` deliver a value. Thus, if stream `x` stops delivering values, stream `d` also stops evaluating and does not need to store any values. △

This example illustrates that even without cycles, specifications in *RTLola+* may not be efficiently monitorable due to different event-based pacings and semantic filters. Streams may need to retain values for future evaluation, resulting in a trace length dependent memory usage. However, not all streams are affected equally, as some may stop evaluating based on their pacing conditions. We can observe that these pacing conditions are met, if the pacing types introduce an synchronous behavior between streams. For instance, in the example above, stream `d` evaluates only when both streams `x` and `y` deliver a value. Therefore, if stream `x` stops delivering values, stream `d` also stops evaluating and does not need to store any values. This synchronous behavior prevents the need for retaining unresolved values, ensuring efficient monitorability for that stream.
Next, we consider specifications with periodic pacings.

Example 4.5.3. Consider the following specification:

```
input a : Bool
output b @a := a.offset(by: 1).defaults(to: false)
output c @1Hz := b.hold(or: false)
```

Here, we do not have any cycles in the dependency graph. Therefore it is well-formed according to Def. 4.2. However, consider the following trace:



We can observe that stream `a` only delivers a value once, s.t. stream `b` also starts its evaluation only once and waits for the future value from stream `a`. However, stream `c` evaluates at a fixed periodic pacing of 1Hz. Therefore, stream `c` continuously accesses the last seen value from stream `b`. Since stream `b` could stay unresolved until T_{max} is reached, stream `c` must store all values until then, leading to an unbounded memory usage. Therefore this specification is not efficiently monitorable. △

The asynchronous pacing types can lead to situations where streams must retain values for future evaluation, resulting in a trace length dependent memory usage. In this case, the stream with the outgoing future offset edge must have event-based pacing. Only then can we use the uncertainty of the evaluation time steps to create such scenarios, since we cannot predict an event-based pacing. In this case, there cannot exist a path to this stream with at least one edge with an asynchronous pacing type, i.e., either a different event-based pacing or a periodic pacing. This combination allows streams to evaluate independently of the future offset stream, leading to unresolved values that need to be stored. Therefore, we need to refine the efficiently monitorable property for *RTLola+* to also exclude such cases.

Theorem 5. *Let* D_Φ *be the dependency graph of a well-formed* RTLola+ *specification* Φ*. If* D_Φ *contains an future offset edge* $(o_x, (t_x, (\text{Offset}, n)), o_y)$ *with* $n > 0$ *and Pace*(o_x) *being event-based, then there must not exist a path to* o_x *that contains at least one edge* $(o_z, (t_z, l), o_w)$ *with Pace*$(o_z) \neq$ *Pace*(o_x)*. Otherwise the streams in this path are unbounded.*

The proof of this theorem follows directly from the examples above. Here, we can observe that the presence of a future offset edge with an event-based pacing can lead to unbounded memory usage if there exists a path to this edge with at least one edge with a different pacing type. This combination allows streams to evaluate independently of the future offset stream, leading to unresolved values that need to be stored. Therefore, we can conclude that specifications with such configurations are not efficiently monitorable. Based on this theorem and the examples above, we can guarantee a bounded memory usage for asynchronously paced specifications in the efficiently monitorable fragment of *RTLola+*. With both theorems combinded, we observe that the efficiently monitorable

property for *RTLola+* is more restrictive than in *Lola* [11], due to the additional challenges introduced by asynchronous pacings. Therefore, we can now summarize the efficiently monitorable property for *RTLola+* as follows:

Corollary 6. *A well-formed* RTLola+ *specification is not efficiently monitorable, if either Thm. 4 or Thm. 5 holds, i.e., if its dependency graph contains a positive cycle or if there exists a future offset edge with an event-based pacing that has a path to it containing at least one edge with a different pacing type.*

This corollary summarizes the efficiently monitorable property for *RTLola+*. It highlights the key conditions that can lead to unbounded memory usage, specifically the presence of positive cycles and certain configurations of future offset edges with event-based pacings. By ensuring that these conditions are not met, we can guarantee that a well-formed *RTLola+* specification is efficiently monitorable, allowing for effective online monitoring with bounded memory requirements.

Remark 4.5.1. *The efficiently monitorable fragment of* RTLola+ *is only relevant for online monitoring scenarios, where we have the uncertainty of the input streams and memory usage is a critical factor. In offline monitoring scenarios, where the entire trace is available beforehand, memory usage is less of a concern. Therefore, the efficiently monitorable property is primarily designed to ensure that specifications can be monitored efficiently in real-time applications.*

4.5.2. Memory Bound

In this section, we analyze the memory usage of a stream from a well-formed *RTLola+* specifications based on Def. 4.8. Here, we differentiate between bounded and unbounded streams according to Def. 4.7. In order to estimate the memory requirements for each stream, we adapt the delay computation from Def. 4.4 to capture all types of unbounded streams.

Definition 4.9 (Memory Delay)

Given a *RTLola+* specification Φ and its corresponding dependency graph D_Φ. We distinguish the *Memory Delay* $\delta(s)$ for each stream $s \in V$ as follows:

$$\delta(s) = \begin{cases} \triangle(s) & \text{if } s \text{ is bounded} \\ +\infty & \text{if } s \text{ is unbounded} \end{cases}$$

The determination of whether a stream is bounded or unbounded follows Def. 4.7 and is extended by Thms. 4 and 5

Based on this memory delay, we can now define the notion of *memory bound* for streams in a specification.

Definition 4.10 (Memory Bound [18])

Given a *RTLola+* specification Φ and its corresponding dependency graph D_Φ. For each stream $s \in V$, we define the *memory bound* $\mu(s)$ as the maximum number of values that need to be stored for stream s to resolve all offset expressions. Formally:

$$\mu(s) = \max\left\{\delta(s), \max\left\{\delta(s') - \omega(s', s) \mid (s', (\_, \_), s) \in E\right\}\right\} + 1$$

Here, the additional 1 accounts for the current value of the stream.

This definition extends the memory bound analysis from *Lola* [18] to account for future offsets in *RTLola+*. The memory bound captures the maximum number of values that need to be stored for a stream to resolve all offset expressions, considering both its own memory delay and the memory delays of streams it depends on through offset edges. This calculation can have two results. If the specification is within the efficiently monitorable fragement, the memory bound will yield a finite value, indicating that the stream requires a bounded amount of memory for its evaluation. Conversely, if the specification is not efficiently monitorable, the memory bound yield an infinite value, indicating that the stream requires an unbounded amount of memory that grows with the trace length. This distinction is crucial for understanding the memory requirements of streams in *RTLola+* specifications. This observation aligns with the definition of bounded and unbounded streams in Def. 4.7.
In order to understand the memory computation better, we will first illustrate the concept of delay with an example.

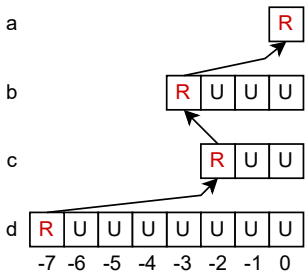
Example 4.5.4. Consider the following specification:

```
input a : Int
output b := a.offset(by: 3).defaults(to: 0)
output c := b.offset(by:-1).defaults(to: 0)
output d := c.offset(by: 5).defaults(to: 0)
```

Here, each stream is bounded. Now, we compute the delay of each stream:

σ	a	b	c	d
$\triangle(\sigma)$	0	3	2	7

This means, when stream `b` evaluates at time step t, it depends on the value of stream `a` at time step $t + 3$. Therefore, stream `b` needs to retain unresolved values for the next 3 time steps to ensure that it can access the required future value of stream `a`. Similarly, stream `c` may remain unresolved for up to 2 time steps, and stream `d` for up to 7 time steps. This behavior can be illustrated with the following memory trace:



The memory trace shows only whether a value is resolved or unresolved at each time step for each stream. Here, the arrows indicate the dependencies between streams at different time steps, highlighting how offsets influence the retention of values. The numbers represent the amount of time steps each value needs to be stored into the past due to the delays introduced by future offsets. △

In the example above, we can observe that the delay of each stream has a direct impact on its memory bound. Streams with higher delays need to retain more unresolved values, leading to increased memory requirements. This observation is crucial for understanding how future offsets influence the memory bound of streams in *RTLola+* specifications.
Here, the delay computation is also sufficient for determining the memory bound. However, in more complex specifications with multiple dependencies and offsets, we need to consider the delays of all streams that a given stream depends on. This is where the memory bound definition becomes essential, as it considers the delays from all relevant streams to compute the overall memory requirements. To illustrate this, we will consider a more complex example.

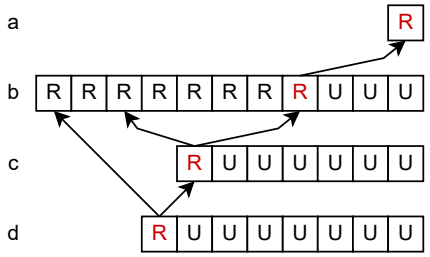
Example 4.5.5. Consider the following specification:

```
input a : Int
output b := a.offset(by: 3).defaults(to: 0)
output c := b.offset(by: -2).defaults(to: 0) + b.offset(by: 3).defaults(to: 0)
output d := c.offset(by: 1).defaults(to: 0) + b.offset(by: -3).defaults(to: 0)
```

Now, we compute the memory delay and memory bound of each stream:

σ	$\delta(\sigma)$	$\mu(\sigma)$
a	0	$\max\{0, \max\{(3-3)\}\}+1=1$
b	3	$\max\{3, \max\{(6-3),(6+2),(7+3)\}\}+1=11$
c	6	$\max\{6, \max\{(7-1)\}\}+1=7$
d	7	$\max\{7, \max\{\}\}+1=8$

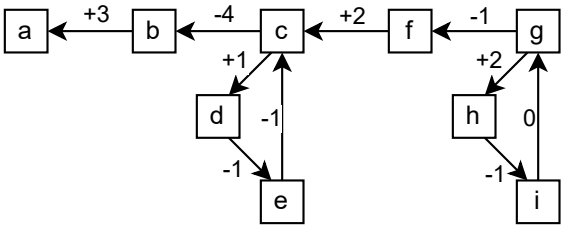
Here, each stream is bounded. Based on this, we can draw the following memory trace:



Here, we can observe that stream `b` has the highest memory bound of 11. This is because stream `b` depends on stream `a` with a future offset of $+3$, leading to a memory delay of 3. Additionally, stream `c` depends on stream `b` with both a future offset of $+3$ and a past offset of -2, resulting in a memory delay of 6. Finally, stream `d` depends on stream `c` with a future offset of $+1$, leading to a memory delay of 7. The memory trace illustrates how each stream retains unresolved values over time, with the arrows indicating dependencies and the numbers representing the time steps each value needs to be stored due to delays. △

In this example, we can see how the memory bound for each stream is influenced by its own memory delay as well as the memory delays of the streams it depends on through offset edges. Stream `b` has the highest memory bound due to its dependencies on both past and future offsets, requiring it to retain a significant number of unresolved values. This analysis highlights the importance of considering all relevant dependencies when calculating the memory bound for streams in *RTLola+* specifications.
Next, we consider specifications with unbounded streams. In this case, the specification is not efficiently monitorable, and the memory bound may yield infinite values for certain streams. These unbounded streams can arise from positive cycles in the dependency graph or from certain configurations of future offset edges with event-based pacings.

Example 4.5.6. Consider the dependency graph from Example 4.4.1.



The corresponding specification is:

```
input a: Int
output b := a.offset(by: +3).defaults(to: 0)
output c := b.offset(by: -4).defaults(to: 0) + d.offset(by: +1).defaults(to: 0)
output d := e.offset(by: -1).defaults(to: 0)
```

```
output e := c.offset(by: -1).defaults(to: 0)
output f := c.offset(by: +2).defaults(to: 0)
output g := f.offset(by: -1).defaults(to: 0) + h.offset(by: +2).defaults(to: 0)
output h := i.offset(by: -1).defaults(to: 0)
output i := g
```

Now, we compute the memory delay and memory bound of each stream:

σ	$\delta(\sigma)$	$\mu(\sigma)$
a	0	$\max\{0, \max\{(3-3)\}\}+1=1$
b	3	$\max\{3, \max\{(1-(-4))\}\}+1=6$
c	1	$\max\{1, \max\{(0-(-1)), (3-2)\}\}+1=2$
d	0	$\max\{0, \max\{(1-1)\}\}+1=1$
e	0	$\max\{0, \max\{(0-(-1))\}\}+1=2$
f	3	$\max\{3, \max\{(\infty-(-1))\}\}+1=\infty$
g	∞	$\max\{\infty, \max\{(\infty-0)\}\}+1=\infty$
h	∞	$\max\{\infty, \max\{(\infty-2)\}\}+1=\infty$
i	∞	$\max\{\infty, \max\{(\infty-(-1))\}\}+1=\infty$

Here, we can observe that streams `g`, `h` and `i` are unbounded, as they are part of a positive cycle in the dependency graph. Consequently, stream `f` is also unbounded, since stream `g` depends on it. The other streams are bounded, with their memory bounds determined by their respective memory delays and dependencies. △

Besides positive cycles, we can also have unbounded streams due to some asynchronous features.

Example 4.5.7. Consider the specification in Example 4.5.3:

```
input a : Bool
output b @a := a.offset(by: 1).defaults(to: false)
output c @1Hz := b.hold(or: false)
```

Now, we compute the memory delay and memory bound of each stream:

σ	$\delta(\sigma)$	$\mu(\sigma)$
a	0	$\max\{0, \max\{(1-1)\}\}+1=1$
b	1	$\max\{1, \max\{(\infty-1)\}\}+1=\infty$
c	∞	$\max\{\infty, \max\{\}\}+1=\infty$

Here, we can observe that streams `b` and `c` are unbounded. Stream `b` has an event-based pacing and depends on stream `a` with a future offset. Since stream `a` may stop delivering values, stream `b` could remain unresolved until T_{max} is reached, leading to unbounded memory usage. Consequently, stream `c` is also unbounded, as it depends on stream `b`. △

These examples illustrate how both positive cycles and certain asynchronous configurations imply unbounded memory usage of streams in *RTLola+* specifications. The memory delay and memory bound calculations help identify which streams require unbounded memory, allowing for better understanding and management of memory usage in monitoring scenarios.
In summary, the memory analysis for *RTLola+* effectively accounts for both past and future offsets. By defining past and future ranges, we can accurately estimate the memory requirements for each stream. This ensures that specifications can be monitored efficiently, even in the presence of complex offset expressions.

Remark 4.5.2. *Sliding windows are not explicitly considered in this memory analysis. These kind of operations need to be treated separately, as they introduce additional memory requirements based on the time window. This behavior and the general correctness of the memory analysis must be further investigated in future work.*

Remark 4.5.3. *The computation of the memory bound is not formally proven to be correct in this work. However, it is based on the memory analysis from* Lola *[18] and adapted to account for future offsets in* RTLola+*. The presented examples illustrate the correctness of the memory bound computation so far. A formal proof of correctness is left for future work.*

Chapter 5

Evaluation

In this chapter, we introduce the implementation of *RTLola+* and discuss the adaptations made to the analysis components to accommodate future offsets. We cover the implementation of the frontend and backend, followed by an evaluation of the memory requirements using a set of benchmark specifications.

5.1. Implementation

In this section, we discuss the implementation details of *RTLola+*, which is based on the *RTLola* framework given in Sect. 3.2.3. We cover both the frontend and backend components, highlighting the key modifications and features introduced to accommodate the future offsets. Both, the frontend and backend, are implemented in Rust.

5.1.1. Frontend

The frontend of *RTLola+* is responsible for parsing and analyzing the specifications. It builds upon the existing *RTLola* frontend with significant modifications in some analysis components to handle future offsets. Particularly, we need to adapt the theoretical results from Chapter 4 into a practical implementation. The main changes are made for the following analysis modes:

- **Dependency Analysis Mode:** Here, we updated the cycle detection algorithm to identify and handle cycles involving future offsets, as discussed in Def. 4.2. This ensures that specifications with ill-defined evaluation models are correctly identified during the analysis phase.
- **Ordered Mode:** We modified the order analysis to account for the delays introduced by future offsets, as outlined in Def. 4.4. This ensures that the evaluation order respects the dependencies created by future offsets, preventing premature

evaluations. The result of this analysis provides several evaluation layer that are used during the monitoring phase.

- **Memory Bound Mode:** The memory analysis was extended to estimate the memory requirements for streams with future offsets, following the approach in Sect. 4.5.2. This includes calculating the exact memory bound for each stream, while differentiating between bounded and unbounded streams.

Each of these modes has been updated to incorporate the considerations for future offsets, ensuring that the frontend can accurately analyze and validate *RTLola+* specifications. After lowering the results of the analysis modes into the mid-level intermediate representation (MIR), the frontend passes the processed specification to the backend for code generation.

5.1.2. Backend

The backend of *RTLola+* is responsible for generating the monitoring code based on the analyzed specifications. Here, we consider another intermediate representation, the *StreamIR* [5], which transforms the MIR from the frontend into a more concrete form that is suitable for code generation. While the MIR captures the declarative semantics of RTLola specifications, the backend translates these into imperative StreamIR programs that operate directly on streams of unbounded length. Each stream is represented as a stateful entity with associated buffers and update mechanisms, allowing for efficient runtime evaluation. The StreamIR has been extended to support future offsets by explicitly differentiating between past and future stream accesses.
Based on these constructs of the StreamIR, the StreamIR interpreter has been adapted to accomodate future offsets. This involes introducing new constructs and operations that handle the delayed evaluation of streams. Here, we need mechanisms for managing unresolved values in the stream buffers and ensuring that future offsets are correctly evaluated based on the current state of the streams. Here, the idea is to delay the evaluation of expressions involving future offsets until the required data becomes available. This involves checking the state of the stream buffers and only performing evaluations when all necessary values are present. The backend also ensures that the generated code adheres to the memory bounds established during the frontend analysis, optimizing buffer sizes and management strategies accordingly.

Remark 5.1.1. *The implementation of the RTLola frontend and the StreamIR interpreter with support for future offsets does not currently support parametrization and sliding windows. These features introduce additional complexity in managing stream states and evaluations, which requires further development and testing to ensure correct functionality. Future work will focus on extending the StreamIR interpreter to fully support these advanced features, allowing for a more comprehensive implementation of* RTLola+ *specifications.*

5.2. Memory Evaluation

In this section, we evaluate the memory requirements of *RTLola+* specifications using a set of benchmark specifications. We assess how future offsets impact memory consumption compared to traditional past-offset-only specifications. We categorize the specifications into two groups: bounded and unbounded *RTLola+* specifications, based on the memory analysis results from the frontend. For each specification, we measure the buffer sizes during runtime monitoring and compare them against the theoretical memory bounds established during the analysis phase.

5.2.1. Bounded *RTLola+* Specifications

We begin by evaluating a set of bounded *RTLola+* specifications, which are designed to operate within fixed memory limits. These specifications utilize future offsets in a manner that does not lead to unbounded memory growth.
First, we consider a simple specification that demonstrates the use of future offsets while maintaining bounded memory usage. Here, we are using the specification from Example 4.5.5 as a benchmark and some random input trace of length 30.

→ Ex. 4.5.5, p. 54

```
input a : Int
output b := a.offset(by: 3).defaults(to: 0)
output c := b.offset(by: -2).defaults(to: 0) + b.offset(by: 3).defaults(to: 0)
output d := c.offset(by: 1).defaults(to: 0) + b.offset(by: -3).defaults(to: 0)
```

During the runtime monitoring of this specification, we track the sum of the buffer sizes for each stream at every time step. Fig. 5.1 illustrates the memory consumption over time. Here, we observe that the memory usage grows over the first 11 time steps up to a maximum of 27 units, after which it stabilizes. This behavior aligns with our expectations based on the memory bound analysis. Particularly, we see that the memory consumption does not exceed the theoretical bound. Also, in the growing phase, the memory usage increases as new unresolved values are added to the buffers due to future offsets. Besides, some resolved values are further stored in the buffers until they are no longer needed for past offsets. Once the buffers reach their maximum required sizes, the memory consumption stabilizes, indicating that the specification is effectively managing its memory usage within the established bounds. This confirms that unresolved values are correctly handled and are always resolved within a constant bound.
To validate the memory analysis, we compare the observed maximum memory usage during runtime monitoring with the theoretical memory bounds calculated during the frontend analysis. The expected memory bounds for each stream are are computed in Example 4.5.5. The total expected memory consumption is $1 + 11 + 7 + 8 = 27$ units, which matches the observed maximum memory usage during runtime monitoring. This confirms that the memory analysis accurately predicts the memory requirements for bounded *RTLola+* specifications with future offsets.

→ Ex. 4.5.5, p. 54

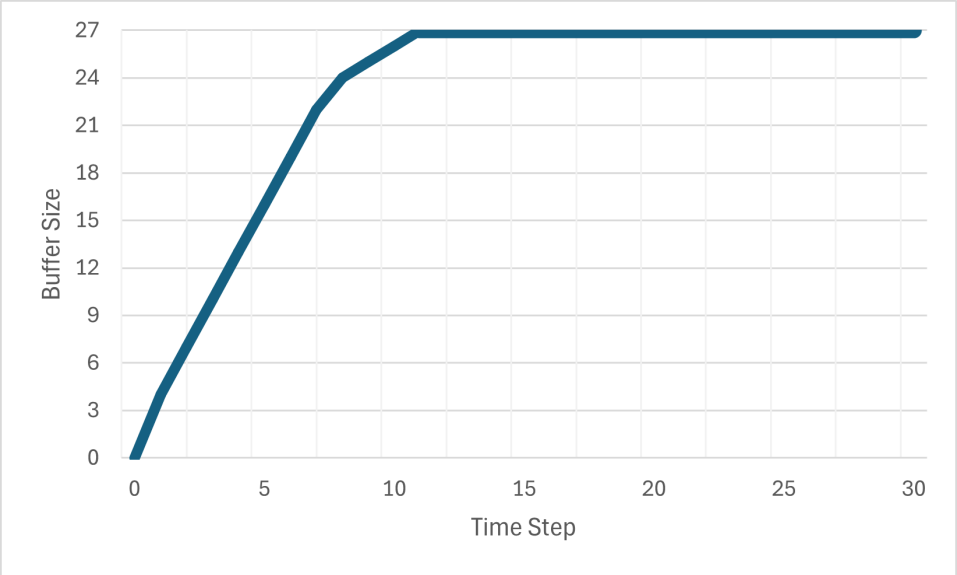


Figure 5.1.: Memory consumption of Example 4.5.5 during runtime monitoring.

Second, we evaluate and compare bounded *RTLola+* specifications with either only past offsets or only future offsets. This comparison aims to highlight the differences in memory consumption patterns between these two types of specifications. We consider two specifications that are structurally similar but differ in their use of offsets:

(a) all-future Specification

```
input a : Int
output b := a.offset(by:+10)
                .defaults(to: 0)
output c := b.offset(by:+20)
                .defaults(to: 0)
output d := c.offset(by:+30)
                .defaults(to: 0)
```

(b) all-past Specification

```
input a : Int
output b := a.offset(by:-10)
                .defaults(to: 0)
output c := b.offset(by:-20)
                .defaults(to: 0)
output d := c.offset(by:-30)
                .defaults(to: 0)
```

The left specification uses only future offsets, while the right specification employs only past offsets. During runtime monitoring, we track the memory consumption for both specifications over time on some random trace of length 100. Fig. 5.3 illustrates the memory usage for both cases. We observe that both specifications exhibit similar memory consumptions, with memory usage growing initially and then stabilizing. However, the future-offset specification shows a higher peak memory usage compared to the past-offset specification. This difference can be attributed to the nature of future offsets, which require maintaining unresolved values in buffers until they are resolved. When multiple future offsets are chained together, this delay sums up to larger buffer sizes. For instance, stream `b` in the all-future specification needs to store unresolved

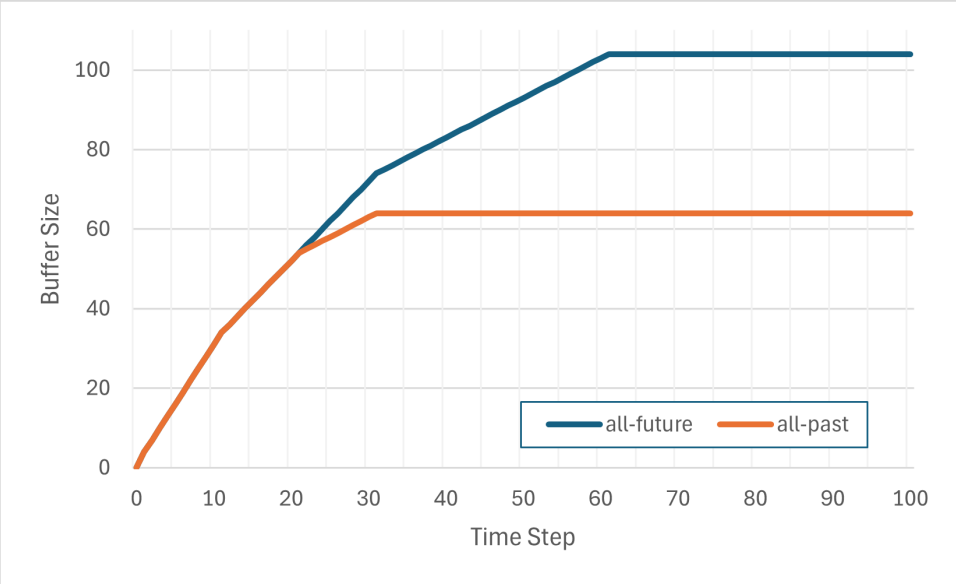


Figure 5.3.: Memory consumption comparison between future-offset and past-offset bounded *RTLola+* specifications during runtime monitoring.

values for 10 time steps until they can be resolved. Since stream `c` accesses stream `b`, it also needs to maintain these unresolved values for 10 time steps and additionally store its own unresolved values for another 20 time steps until they can be resolved. The results is a total of 30 time steps of unresolved values for stream `c`. Consequently, the buffer sizes grow significantly for streams that depend on multiple future offsets. In contrast, the past-offset specification can resolve values immediately based on previously stored data, since past offsets refer to already available values. This leads to a lower memory consumption overall.

Despite this difference, both specifications remain within their theoretical memory bounds, confirming the effectiveness of the memory analysis for both types of offset usage. The expected memory bounds for each stream in both specifications are as follows:

σ	a	b	c	d
$\mu(\sigma)$	1	11	31	61

(a) all-future Memory

σ	a	b	c	d
$\mu(\sigma)$	11	21	31	1

(b) all-past Memory

The total expected memory consumption for the all-future specification is $1 + 11 + 31 + 61 = 104$ units, while for the all-past specification, it is $11 + 21 + 31 + 1 = 64$ units. These values align with the observed maximum memory usage during runtime

monitoring, confirming the accuracy of the memory analysis for both future and past offset specifications.

Last, we evaluate two bounded *RTLola+* specifications that are seeking the same monitoring goal but use different combinations of past and future offsets. This comparison aims to illustrate how the choice of offset types can impact memory consumption while achieving the same functional outcome. We consider the following two specifications:

(a) `future` Specification:

```
input a: Int
input b: Int
input c: Int
output d := a.offset(by: +2).defaults(to: 0)
              + b.offset(by: -3).defaults(to: 0)
              + b.offset(by: +5).defaults(to: 0)
              + c.offset(by: +9).defaults(to: 0)
```

(b) `past` Specification:

```
input a: Int
input b: Int
input c: Int
output d := a.offset(by: -7).defaults(to: 0)
              + b.offset(by: -12).defaults(to: 0)
              + b.offset(by: -4).defaults(to: 0)
              + c
```

Both specifications compute the same output stream `d` based on the input streams `a`, `b`, and `c`. The `future` specification uses future offsets, delaying the evaluation, while the `past` specification translates these delays into past offsets by subtracting the maximum future offset. During runtime monitoring on some random input trace, both specifications produce identical output values for stream `d`, though the `past` specification's output is delayed by 9 time steps.
Fig. 5.5 illustrates the memory consumption for both specifications. The `future` specification exhibits higher memory usage due to maintaining unresolved values for future offsets alongside resolved values for past offsets. In contrast, the `past` specification shows lower memory consumption as it resolves values immediately based on past data. This highlights the memory efficiency trade-offs between future and past offsets while achieving the same monitoring goal. Both specifications remain within their theoretical memory bounds, validating the memory analysis accuracy.

In summary, the evaluation of bounded *RTLola+* specifications demonstrates that future offsets can lead to higher memory consumption compared to past offsets, primarily due to the need to maintain unresolved values in buffers. However, both types of specifications can effectively manage their memory usage within established theoretical bounds, confirming the accuracy of the memory analysis. The choice between future and past offsets should consider the trade-offs in memory consumption while achieving the desired monitoring functionality.

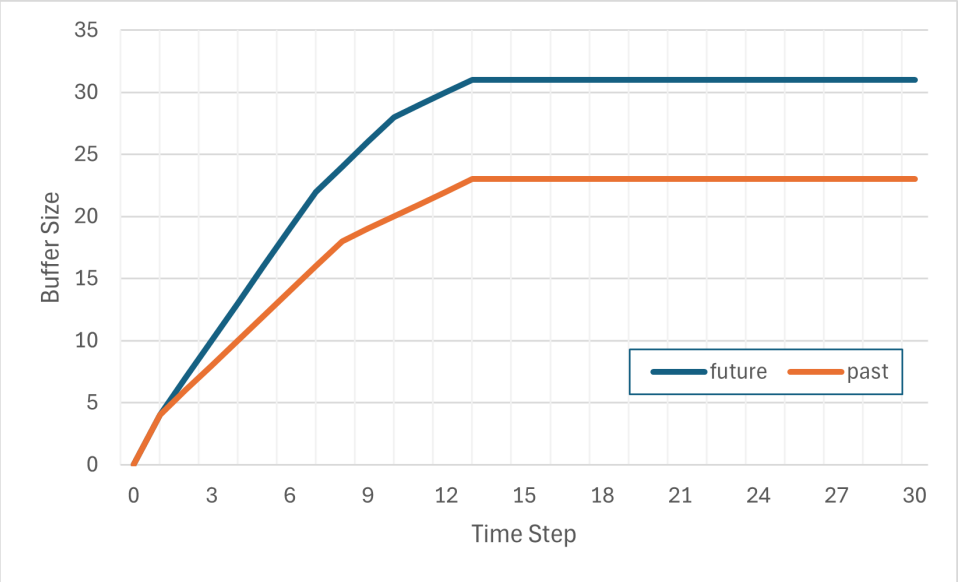


Figure 5.5.: Memory consumption comparison between `future` and `past` bounded *RTLola+* specifications during runtime monitoring.

5.2.2. Unbounded *RTLola+* Specifications

Next, we evaluate a set of unbounded *RTLola+* specifications, which inherently require dynamic memory allocation due to their design. Unlike bounded specifications, which we can translate to equivalent past-offset-only specifications, unbounded specifications always involve future offsets that lead to unbounded memory growth. This unbounded behavior introduces complexities that are not present in traditional *RTLola* specifications and cannot be expressed using only past offsets. Here, we assess how these specifications behave during runtime monitoring, particularly focusing on their memory consumption patterns.
First, we consider an unbounded specification that utilizes future offsets in a manner that leads to unbounded memory consumption.

```
input a : Int
output b @a := a + c.offset(by: +1).defaults(to: 0)
output c @a := b.offset(by: +9).defaults(to: 0)
```

During runtime monitoring of this specification on some random input trace of length 1000, we track the memory consumption over time. Fig. 5.6 illustrates the memory usage. Here, we observe that the memory consumption grows continuously without stabilizing, indicating unbounded growth. This behavior is expected due to the positive cycle created by the future offsets in the specification. Each new input value leads to additional unresolved values being added to the buffers of streams `b` and `c`. This creates a situation where the buffers cannot be resolved in a timely manner, leading to continuous

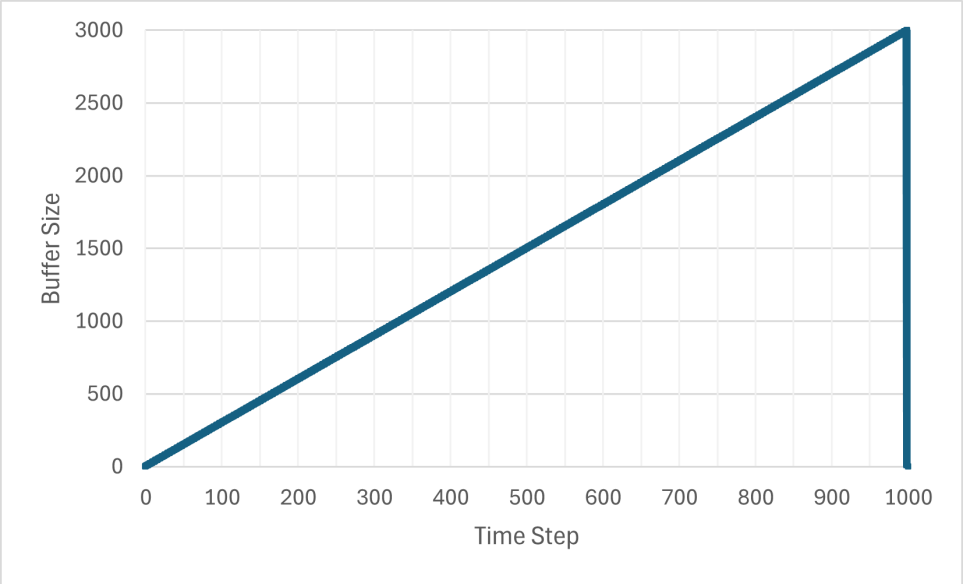


Figure 5.6.: Memory consumption of unbounded *RTLola+* specification during runtime monitoring.

growth in memory usage. However, this linear growth pattern interrupts at the end of the input trace. At this point, the future offsets access their default values and therefore resolve all remaining unresolved values in the buffers. This results in a drop in memory consumption, as all buffered values are cleared.
Moreover, the unbounded nature of this specification highlights the challenges associated with managing memory in the presence of future offsets. As a result, the memory consumption continues to increase as more input values are processed, demonstrating the limitations of such specifications in practical monitoring scenarios.

However, not all unbounded *RTLola+* specifications exhibit continuous memory growth. It is also possible to design unbounded specifications which allows for clearing the buffers of streams at specific points in time. Consider the following unbounded specification, where we count and output the number of time steps until the next reset occurs:

```
input reset: Bool
output x @reset := if reset then 0 else x.offset(by: 1).defaults(to: 0) + 1
```

Here, the evaluation of stream `x` depends on the value of the input stream `reset`. When `reset` is false, stream `x` refers to its next instance, effectively creating a positve cycle that would typically lead to unbounded memory growth. However, when `reset` is true, the stream `x` evaluates to a resolved constant value of `0`. Therefore, this evaluation can resolve all previously unresolved values in the buffer of stream `x`, effectively clearing the buffer and resetting the memory usage for that stream.

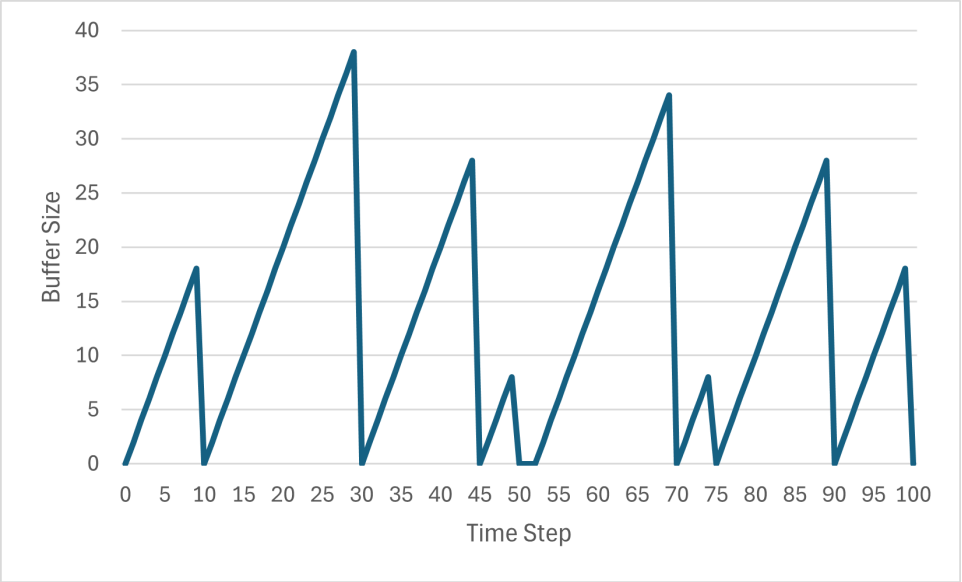


Figure 5.7.: Memory consumption of unbounded *RTLola+* specification with resets during runtime monitoring.

In this case, we consider an input trace of length 100 with 8 resets distributed throughout the trace, where `reset` is being true at those points. At all other time steps, `reset` is false. During runtime monitoring of this specification on this trace, we track the memory consumption over time. Fig. 5.7 illustrates the memory usage pattern. Here, we observe that the memory consumption grows when `reset` is false, as unresolved values accumulate in the buffer of stream `x`. However, whenever `reset` is true, there is a drop in memory consumption, indicating that the buffer has been cleared. This pattern repeats throughout the monitoring process, demonstrating how the specification reset mechanism effectively manages memory usage even in an unbounded specification.
This shows that unbounded *RTLola+* specifications can provide a practical means of controlling memory consumption, preventing unbounded growth and ensuring more efficient monitoring. Therefore, if we can guarantee a maximum distance between two resets in the input trace, we can establish a memory bound for the specification. In this case, the memory bound depends on the maximum number of time steps between two resets, as this determines the maximum number of unresolved values that can accumulate in the buffer of the stream. For instance, we can define the reset as an output stream with periodic pacing to ensure a fixed interval between resets. In this case, the output stream `x` needs to access the reset stream asynchronously, as the resets may not align with the pacing of stream `x`. Therefore, stream `x` clears its buffer frequently, implying an unbounded specification with an practically bounded memory consumption.
Finally, we evaluate an unbounded *RTLola+* specification that operates asynchronously, meaning that the input streams have different pacings. This asynchrony can lead to

different memory consumptions, as the timing of input values can significantly impact the resolution of buffered values. Consider the following unbounded specification from Example 4.5.2:

```
input x: Bool
input y: Bool
output a @x := x.offset(by: 4).defaults(to:false)
output b @y := a.hold(or: false)
output c @(x||y) := a.hold(or: false)
output d @(x&&y) := a.hold(or: false)
```

This specification is asynchronous due to the different pacings of the output streams. We track the buffer size of this specification during runtime monitoring. Fig. 5.8 illustrates the memory consumption over time for some input trace. Here, we observe that the total buffer size grows over the first 6 time steps up to a maximum of 20 units, after which it stabilizes. This behavior indicates that both input streams `x` and `y` receive their values synchronously during the first 20 time steps. After this point, the buffer size grows linearly, indicating that the input stream `x` does not receive any new values, while `y` continues to receive values asynchronously. This leads to unresolved values accumulating in the buffers of streams `b` and `c`, resulting in continuous memory growth. After the input stream `x` starts receiving new values again and stream `a` resolves its values, the total buffer size drops again to 20 units. The same pattern repeats at time step 68 and 79. In general, we observe that the total buffer size grows whenever the input stream `x` does not receive any new values, while the groth depends on the length of these pauses. On the other hand, whenever the input stream `y` does not receive any new values, the total buffer size drops, as all streams depending on stream `a` can resolve their values and do not produce new unresolved values. Therefore, these values can be dropped from the buffers. After the input stream `y` starts receiving new values again, the total buffer size grows again up to 20 units.
Overall, we notice that the memory consumption of this asynchronous unbounded *RTLola+* specification remains constant as long as both input streams receive their values synchronously. Whenever only stream `y` receives values, the memory consumption grows. Otherwise, when only stream `x` receives values, the memory consumption drops.
In summary, the evaluation of unbounded *RTLola+* specifications highlights the challenges associated with managing memory in the presence of future offsets. While some specifications may lead to continuous memory growth, other specifications can controll their memory consumption by dropping buffered values at specific points in time, ensuring more efficient memory usage. Overall, these evaluations demonstrate the importance of careful specification design and the potential benefits of incorporating memory management strategies in *RTLola+* specifications.

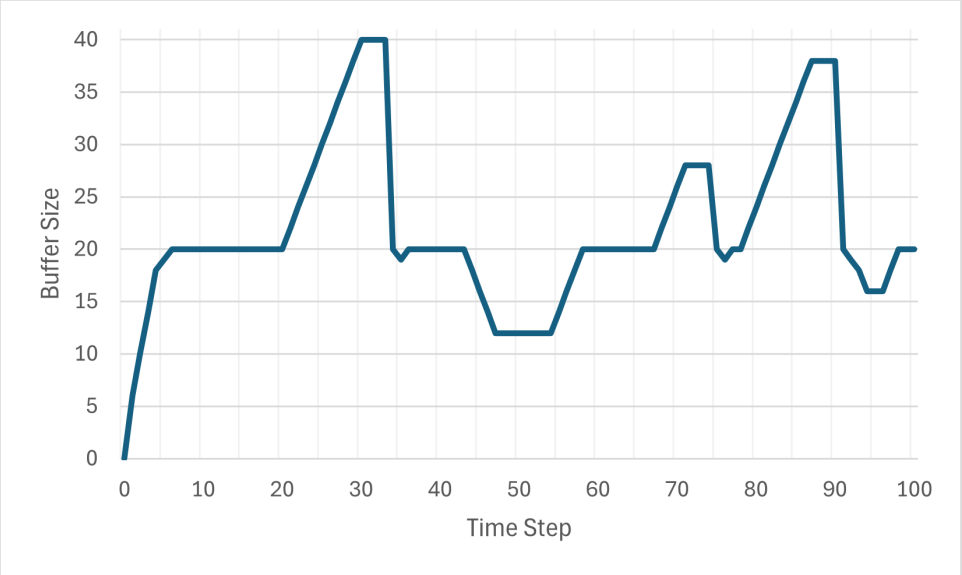


Figure 5.8.: Memory consumption of unbounded asynchronous *RTLola+* specification during runtime monitoring.

Chapter 6

Conclusion and Future Work

6.1. Conclusion

This thesis introduces *RTLola+*, an extension of the stream-based specification language *RTLola* which incorporates the concept of future offsets from *Lola*. The addition of future offsets enhances the expressiveness of the language, allowing specifications to reference values that occur at later points in time. This extension closes the gap in the original design of *RTLola*, which was limited to past and present references.

The increased expressiveness of *RTLola+* is evident in several ways. Specifications can now introduce artificial delays in the evaluation of streams. This enables the modeling of systems where we want to react to events after a certain delay. Furthermore, the extension makes it possible to express temporal operators from Linear Temporal Logic (LTL), such as eventually or globally, thereby bridging the gap between stream-based monitoring and temporal logics. These capabilities allow *RTLola+* to capture a wider range of temporal requirements than *RTLola*.

To support these new features, we defined the formal semantics of *RTLola+*, ensuring that specifications involving future offsets are well-defined and unambiguous. Particular attention was given to the well-formedness criterion, which we address through a dependency analysis that guarantees the absence of evaluation deadlocks in *RTLola+* specifications. This ensures that specifications can be evaluated consistently without leading to undefined behaviors. Furthermore, we extend the order analysis to accomodate future offsets, allowing us to determine a correct evaluation order for streams in *RTLola+* specifications. In addition, we adapt the memory analysis to account for the storage requirements introduced by future offsets. Here, we want to ensure that the extended language remains efficiently monitorable. Therefore, we classify specifications based on their memory requirements, distinguishing between those that can be monitored with bounded memory and those that require unbounded memory. Bounded Specifications can be monitored with a fixed amount of memory regardless of the length

of the trace, while unbounded specifications require storing unresolved values until the corresponding future data becomes available. To handle this, we introduced the distinction between resolved and unresolved states for stream values in *RTLola+*, providing a mechanism to manage future offsets while maintaining correctness. By implementing these semantics, we provide a theoretical foundation for the extended language. Thereby, we adapt the monitoring algorithm accordingly to efficiently handle future offsets, ensuring that the extended language remains practical for real-time monitoring applications.

In order to validate the theoretical foundations of the memory analysis of *RTLola+*, we evaluated the implementation of *RTLola+* using a set of benchmark specifications. These benchmarks demonstrated the practical applicability of the memory analysis, confirming that the classification into bounded and unbounded specifications reflects the memory usage during monitoring. The evaluation showed that *RTLola+* can effectively monitor specifications with future offsets while managing memory usage according to the theoretical predictions.

In summary, extending *RTLola* to *RTLola+* with future offsets provides a trade-off between expressiveness and memory consumption. Although future offsets enable *RTLola+* to specify mixed temporal properties, they also present memory management challenges. Not all specifications with future offsets can be monitored with bounded memory, highlighting the importance of careful consideration when designing specifications. However, *RTLola+* specifications with unbounded memory requirements can still be useful in scenarios where the trace length is limited or the memory consumption is guaranteed to remain manageable.

6.2. Future Work

Although we provided a theoretical foundation for *RTLola+*, we left some questions open for future research.

The basis of this extension is a fragement of *RTLola*, where we exclude features such as `spawn`-clauses, `close`-clauses and parameterized streams. Therefore, the combination of these feature with future offsets needs to be investigated, as they introduce additional complexity to the language semantics and monitoring process. Particularly, these features lead to multiple concurrent instances and dynamic creation of streams based on runtime conditions. This investigation involves revisiting all of the analysis steps in *RTLola+* to ensure that they are correct and efficient in the presence of these additional features.

Besides that, the correctness of the order and memory analysis for *RTLola+* is mainly based on intuition and informal arguments. Formal proofs of correctness will strengthen the theoretical foundations of *RTLola+* and provide stronger guarantees about the behavior of monitors generated from *RTLola+* specifications. Additionally, the order and memory analyses need to be extended to account for the use of sliding windows with

future offsets. This involves analysing how future offsets interact with sliding windows and establishing the impact on memory usage and evaluation order.
Additionally, we can further investigate the behavior of periodic stream in combination with asynchronous operators and future offsets. Here, the goal is to identify tighter criteria for well-formed specifications. Particularly, there are cases in *RTLola+* for asynchronous cycles which are forbidden by the current well-formedness criterion, but can be allowed under certain conditions. By this investigation, we analyse the implications on expressiveness and implementability.
Furthermore, exploring the expressiveness of *RTLola+* compared to other temporal logics provides insight into its capabilities and limitations. This involves formal comparisons with logics such as Metric Temporal Logic (MTL) [25] or Signal Temporal Logic (STL) [27]. The goal is to identify properties that can be expressed in *RTLola+* but not in these logics and vice versa. Based on these comparisons, we can identify potential extensions or modifications to enhance the expressiveness of *RTLola+*. Additionally, we can compare the expressiveness of *RTLola+* with that of other stream-based specification languages as *Striver* [20] and *TeSSLa* [9]. This comparison categorizes these languages based on their expressiveness, identifying the unique features and capabilities of each language, as well as their limitations.

Appendix A

Appendix

A.1. Inference Rules For Expressions

The inference rules for evaluating expressions are based on the semantics of stream expressions in *RTLola* as defined in [5] and are extended to support the new offset expressions introduced in *RTLola+*.

A.1.1. Inference Rules For *RTLola*

The inference rules for evaluating expressions in *RTLola* are defined as follows:

EVAL-EXPR-VAL

$$\frac{v \in \mathbb{V}}{\texttt{Constant}(v) \Downarrow_\omega^t v}$$

EVAL-EXPR-SYN

$$\frac{\omega(\text{sid})(t) = v}{\texttt{Sync}(\text{sid}) \Downarrow_\omega^t v}$$

EVAL-EXPR-FUNCTION

$$\frac{p_1 \Downarrow_\omega^t v_1 \dots p_n \Downarrow_\omega^t v_n \quad f(v_1, \dots, v_n) = v_{res}}{\texttt{Function}(f, p_1, \dots, p_n) \Downarrow_\omega^t v_{res}}$$

EVAL-EXPR-OFF

$$\frac{\text{prefix} = \texttt{Prefix}(\omega, \text{sid}, t) \quad |\text{prefix}| > \text{off} \quad \text{prefix}[|\text{prefix}| - \text{off}] = (t', v)}{\texttt{Offset}(\text{sid}, \text{off}) \Downarrow_\omega^t v}$$

EVAL-EXPR-OFF-DFT

$$\frac{\text{prefix} = \texttt{Prefix}(\omega, \text{sid}, t) \quad |\text{prefix}| \leqslant \text{off}}{\texttt{Offset}(\text{sid}, \text{off}) \Downarrow_\omega^t \bot}$$

EVAL-EXPR-HOLD

$$\frac{\mathit{prefix} = \texttt{Prefix}(\omega, \mathit{sid}, t) \quad \mathit{prefix} = \mathit{prefix}' \cdot (t', v)}{\texttt{Hold}(\mathit{sid}) \Downarrow^t_\omega v}$$

EVAL-EXPR-HOLD-DFT

$$\frac{\mathit{prefix} = \texttt{Prefix}(\omega, \mathit{sid}, t) \quad \mathit{prefix} = \epsilon}{\texttt{Hold}(\mathit{sid}) \Downarrow^t_\omega \bot}$$

EVAL-EXPR-NO-DFT

$$\frac{\mathit{expr} \Downarrow^t_\omega v \quad v \in \mathbb{V}}{\texttt{Default}(\mathit{expr}, \mathrm{dft}) \Downarrow^t_\omega v}$$

EVAL-EXPR-DFT

$$\frac{\mathit{expr} \Downarrow^t_\omega \bot \quad \mathrm{dft} \Downarrow^t_\omega v}{\texttt{Default}(\mathit{expr}, \mathrm{dft}) \Downarrow^t_\omega v}$$

EVAL-EXPR-AGGREGATE

$$\frac{\mathit{prefix} = \texttt{Prefix}(\omega, \mathit{sid}, t) \quad \mathit{window} = \texttt{Window}(\omega, \mathit{prefix}, \omega(t) - \mathrm{dur}) \quad f_a(\mathit{window} = v)}{\texttt{Aggregate}(\mathit{sid}, \mathrm{dur}, f_a) \Downarrow^t_\omega v}$$

A.1.2. Additional Inference Rules For *RTLola+*

The inference rules for evaluating offset expressions in *RTLola+* are defined as follows:

EVAL-EXPR-PAST-OFF

$$\frac{\mathit{prefix} = \texttt{Prefix}(\omega, \mathit{sid}, t) \quad |\mathit{prefix}| > \mathrm{off} \quad \mathit{prefix}[|\mathit{prefix}| - \mathrm{off}] = (t', v)}{\texttt{PastOffset}(\mathit{sid}, \mathrm{off}) \Downarrow^t_\omega v}$$

EVAL-EXPR-PAST-OFF-DFT

$$\frac{\mathit{prefix} = \texttt{Prefix}(\omega, \mathit{sid}, t) \quad |\mathit{prefix}| \leqslant \mathrm{off}}{\texttt{PastOffset}(\mathit{sid}, \mathrm{off}) \Downarrow^t_\omega \bot}$$

EVAL-EXPR-FUTURE-OFF

$$\frac{\mathit{suffix} = \texttt{Suffix}(\omega, \mathit{sid}, t) \quad |\mathit{suffix}| > \mathrm{off} \quad \mathit{suffix}[\mathrm{off}] = (t', v)}{\texttt{FutureOffset}(\mathit{sid}, \mathrm{off}) \Downarrow^t_\omega v}$$

EVAL-EXPR-FUTURE-OFF-DFT

$$\frac{\mathit{suffix} = \texttt{Suffix}(\omega, \mathit{sid}, t) \quad |\mathit{suffix}| \leqslant \mathrm{off}}{\texttt{FutureOffset}(\mathit{sid}, \mathrm{off}) \Downarrow^t_\omega \bot}$$

A.2. Functions For Inference Rules

A.2.1. Functions For *RTLola*

Prefix

$$\texttt{Prefix} : \mathbb{W} \times \mathrm{ID}^* \times \mathrm{Time} \to (\mathrm{Time}, \mathbb{V})^*$$

$$\texttt{Prefix}(\omega, \mathrm{sid}, t)$$

$$:= \begin{cases} (t, v) & \text{if } t = 0 \wedge \omega(\mathrm{sid})(t) = v \wedge v \in \mathbb{V} \\ \epsilon & \text{if } t = 0 \wedge \omega(\mathrm{sid})(t) = \bot \\ \texttt{Prefix}(\omega, \mathrm{sid}, t-1) \cdot (t, v) & \text{if } \omega(\mathrm{sid})(t) = v \wedge v \in \mathbb{V} \\ \texttt{Prefix}(\omega, \mathrm{sid}, t-1) & \text{if } \omega(\mathrm{sid})(t) = \bot \end{cases}$$

Window

$$\texttt{Window} : \mathbb{W} \times (\mathrm{Time}, \mathbb{V})^* \times \mathrm{Time} \to (\mathrm{Time}, \mathbb{V})^*$$

$$\texttt{Window}(\omega, \mathrm{pre}, t_{start})$$

$$:= \begin{cases} \texttt{Window}(\omega, \mathrm{pre}', t_{start}) \cdot (t, v) & \text{if } \mathrm{pre} = \mathrm{pre}' \cdot (t, v) \wedge \omega(t) > t_{start} \\ \epsilon & \text{otherwise} \end{cases}$$

A.2.2. Additional Function For *RTLola+*

Suffix

$$\texttt{Suffix} : \mathbb{W} \times \mathrm{ID}^* \times \mathrm{Time} \to (\mathrm{Time}, \mathbb{V})^*$$

$$\texttt{Suffix}(\omega, \mathrm{sid}, t)$$

$$:= \begin{cases} (t, v) & \text{if } t = T_{max} \wedge \omega(\mathrm{sid})(t) = v \wedge v \in \mathbb{V} \\ \epsilon & \text{if } t = T_{max} \wedge \omega(\mathrm{sid})(t) = \bot \\ (t, v) \cdot \texttt{Suffix}(\omega, \mathrm{sid}, t+1) & \text{if } \omega(\mathrm{sid})(t) = v \wedge v \in \mathbb{V} \\ \texttt{Suffix}(\omega, \mathrm{sid}, t+1) & \text{if } \omega(\mathrm{sid})(t) = \bot \end{cases}$$

Bibliography

[1] Florian-Michael Adolf et al. "Stream Runtime Monitoring on UAS". In: *Runtime Verification*. Springer International Publishing, 2017, pp. 33–49. ISBN: 97833196753-12. DOI: 10.1007/978-3-319-67531-2_3. URL: http://dx.doi.org/10.1007/978-3-319-67531-2_3.

[2] Tzanis Anevlavis et al. "Evrostos: the rLTL verifier". In: *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*. HSCC '19. ACM, Apr. 2019, pp. 218–223. DOI: 10.1145/3302504.3311812. URL: http://dx.doi.org/10.1145/3302504.3311812.

[3] Jan Baumeister, Bernd Finkbeiner, and Frederik Scheerer. *Active Monitoring with RTLola: A Specification-Guided Scheduling Approach*. 2025. arXiv: 2507.20615 [cs.LO]. URL: https://arxiv.org/abs/2507.20615.

[4] Jan Baumeister et al. "A Tutorial on Stream-Based Monitoring". In: *Formal Methods*. Springer Nature Switzerland, Sept. 2024, pp. 624–648. ISBN: 9783031711770. DOI: 10.1007/978-3-031-71177-0_33. URL: http://dx.doi.org/10.1007/978-3-031-71177-0_33.

[5] Jan Baumeister et al. *An Intermediate Program Representation for Optimizing Stream-Based Languages*. 2025. arXiv: 2504.21458 [cs.LO]. URL: https://arxiv.org/abs/2504.21458.

[6] Jan Baumeister et al. "Real-Time Visualization of Stream-Based Monitoring Data". In: *Runtime Verification*. Springer International Publishing, 2022, pp. 325–335. ISBN: 9783031171963. DOI: 10.1007/978-3-031-17196-3_21. URL: http://dx.doi.org/10.1007/978-3-031-17196-3_21.

[7] Jan Baumeister et al. "RTLola Cleared for Take-Off: Monitoring Autonomous Aircraft". In: *Computer Aided Verification*. Springer International Publishing, 2020, pp. 28–39. ISBN: 9783030532918. DOI: 10.1007/978-3-030-53291-8_3. URL: http://dx.doi.org/10.1007/978-3-030-53291-8_3.

[8] Jan Baumeister et al. *Stream-Based Monitoring of Algorithmic Fairness*. 2025. arXiv: 2501.18331 [cs.LG]. URL: https://arxiv.org/abs/2501.18331.

[9] Lukas Convent et al. "Hardware-Based Runtime Verification with Embedded Tracing Units and Stream Processing". In: *Runtime Verification*. Springer International Publishing, 2018, pp. 43–63. ISBN: 9783030037697. DOI: 10.1007/978-3-030-03769-7_5. URL: http://dx.doi.org/10.1007/978-3-030-03769-7_5.

[10] Lukas Convent et al. "TeSSLa: Temporal Stream-Based Specification Language". In: *Formal Methods: Foundations and Applications*. Springer International Publishing, 2018, pp. 144–162. ISBN: 9783030030445. DOI: 10.1007/978-3-030-03044-5_10. URL: http://dx.doi.org/10.1007/978-3-030-03044-5_10.

[11] B. D'Angelo et al. "LOLA: runtime monitoring of synchronous systems". In: *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*. 2005, pp. 166–174. DOI: 10.1109/TIME.2005.26.

[12] Doron Drusinsky. "The Temporal Rover and the ATG Rover". In: *SPIN Model Checking and Software Verification*. Springer Berlin Heidelberg, 2000, pp. 323–330. ISBN: 9783540452973. DOI: 10.1007/10722468_19. URL: http://dx.doi.org/10.1007/10722468_19.

[13] Yliès Falcone and César Sánchez. "Introduction to the special issue on runtime verification". In: *Formal Methods in System Design* 53.1 (June 2018), pp. 1–5. ISSN: 1572-8102. DOI: 10.1007/s10703-018-0320-4. URL: http://dx.doi.org/10.1007/s10703-018-0320-4.

[14] Peter Faymonville et al. "A Stream-Based Specification Language for Network Monitoring". In: *Runtime Verification*. Springer International Publishing, 2016, pp. 152–168. ISBN: 9783319469829. DOI: 10.1007/978-3-319-46982-9_10. URL: http://dx.doi.org/10.1007/978-3-319-46982-9_10.

[15] Peter Faymonville et al. *Real-time Stream-based Monitoring*. 2019. arXiv: 1711.03829 [cs.LO]. URL: https://arxiv.org/abs/1711.03829.

[16] Peter Faymonville et al. "StreamLAB: Stream-based Monitoring of Cyber-Physical Systems". In: *Computer Aided Verification*. Springer International Publishing, 2019, pp. 421–431. ISBN: 9783030255404. DOI: 10.1007/978-3-030-25540-4_24. URL: http://dx.doi.org/10.1007/978-3-030-25540-4_24.

[17] Bernd Finkbeiner and Lars Kuhtz. "Monitor Circuits for LTL with Bounded and Unbounded Future". In: *Runtime Verification*. Springer Berlin Heidelberg, 2009, pp. 60–75. ISBN: 9783642046940. DOI: 10.1007/978-3-642-04694-0_5. URL: http://dx.doi.org/10.1007/978-3-642-04694-0_5.

[18] Bernd Finkbeiner et al. "Verified Rust Monitors for Lola Specifications". In: Oct. 2020, pp. 431–450. ISBN: 978-3-030-60507-0. DOI: 10.1007/978-3-030-60508-7_24.

[19] Felipe Gorostiaga and César Sánchez. "Stream runtime verification of real-time event streams with the Striver language". In: *Int. J. Softw. Tools Technol. Transf.* 23.2 (Apr. 2021), pp. 157–183. ISSN: 1433-2779. DOI: 10.1007/s10009-021-00605-3. URL: https://doi.org/10.1007/s10009-021-00605-3.

[20] Felipe Gorostiaga and César Sánchez. "Striver: Stream Runtime Verification for Real-Time Event-Streams". In: *Proc. of the 18th Int'l Conf. on Runtime Verification (RV'18)*. Vol. 11237. LNCS. Springer, 2018, pp. 282–298. DOI: 10.1007/978-3-030-03769-7_16. URL: https://link.springer.com/chapter/10.1007/978-3-030-03769-7_16.

[21] N. Halbwachs et al. "The synchronous data flow programming language LUSTRE". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320. DOI: 10.1109/5.97300.

[22] Klaus Havelund and Allen Goldberg. "Verify Your Runs". In: *Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*. Ed. by Bertrand Meyer and Jim Woodcock. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 374–383. ISBN: 978-3-540-69149-5. DOI: 10.1007/978-3-540-69149-5_40. URL: https://doi.org/10.1007/978-3-540-69149-5_40.

[23] Klaus Havelund and Grigore Roşu. "An Overview of the Runtime Verification Tool Java PathExplorer". In: *Formal Methods in System Design* 24.2 (Mar. 2004), pp. 189–215. ISSN: 0925-9856. DOI: 10.1023/b:form.0000017721.39909.4b. URL: http://dx.doi.org/10.1023/B:FORM.0000017721.39909.4b.

[24] Stefan Jakšić et al. "From signal temporal logic to FPGA monitors". In: *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. 2015, pp. 218–227. DOI: 10.1109/MEMCOD.2015.7340489.

[25] Ron Koymans. "Specifying real-time properties with metric temporal logic". In: *Real-Time Syst.* 2.4 (Oct. 1990), pp. 255–299. ISSN: 0922-6443. DOI: 10.1007/BF01995674. URL: https://doi.org/10.1007/BF01995674.

[26] Martin Leucker and Christian Schallhart. "A brief account of runtime verification". In: *The Journal of Logic and Algebraic Programming* 78.5 (2009). The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07), pp. 293–303. ISSN: 1567-8326. DOI: https://doi.org/10.1016/j.jlap.2008.08.004. URL: https://www.sciencedirect.com/science/article/pii/S1567832608000775.

[27] Oded Maler and Dejan Nickovic. "Monitoring Temporal Properties of Continuous Signals". In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Springer Berlin Heidelberg, 2004, pp. 152–166. ISBN: 9783540302063. DOI: 10.1007/978-3-540-30206-3_12. URL: http://dx.doi.org/10.1007/978-3-540-30206-3_12.

[28] Corto Mascle et al. "From LTL to rLTL monitoring: improved monitorability through robust semantics". In: *Formal Methods in System Design* 59.1–3 (Dec. 2021), pp. 170–204. ISSN: 1572-8102. DOI: 10.1007/s10703-022-00398-4. URL: http://dx.doi.org/10.1007/s10703-022-00398-4.

[29] Wolf Dieter Pietruszka and Michael Glöckler. "Simulation unter MATLAB®". In: *MATLAB® und Simulink® in der Ingenieurpraxis*. Springer Fachmedien Wiesbaden, 2021, pp. 265–374. ISBN: 9783658297404. DOI: 10.1007/978-3-658-29740-4_5. URL: http://dx.doi.org/10.1007/978-3-658-29740-4_5.

[30] Amir Pnueli. "The temporal logic of programs". In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[31] Akshay Rajhans et al. "Specification and Runtime Verification of Temporal Assessments in Simulink". In: *Runtime Verification*. Springer International Publishing, 2021, pp. 288–296. ISBN: 9783030884949. DOI: 10.1007/978-3-030-88494-9_17. URL: http://dx.doi.org/10.1007/978-3-030-88494-9_17.

[32] Torben Scheffel. "Expressiveness and Complexity of Stream-based Specification Languages". PhD thesis. Universität zu Lübeck, 2020.

[33] Johann Schumann, Patrick Moosbrugger, and Kristin Y. Rozier. "R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems". In: *Runtime Verification*. Springer International Publishing, 2015, pp. 233–249. ISBN: 9783319238203. DOI: 10.1007/978-3-319-23820-3_15. URL: http://dx.doi.org/10.1007/978-3-319-23820-3_15.

[34] Maximilian Schwenger. "Statically-analyzed stream monitoring for cyber-physical Systems". PhD thesis. 2022. DOI: http://dx.doi.org/10.22028/D291-37014.

[35] Paulo Tabuada and Daniel Neider. "Robust Linear Temporal Logic". In: *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*. Ed. by Jean-Marc Talbot and Laurent Regnier. Vol. 62. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016, 10:1–10:21. ISBN: 978-3-95977-022-4. DOI: 10.4230/LIPIcs.CSL.2016.10. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CSL.2016.10.

[36] Christoph Torens et al. "Towards Intelligent System Health Management using Runtime Monitoring". In: *AIAA Information Systems-AIAA Infotech @ Aerospace*. American Institute of Aeronautics and Astronautics, Jan. 2017. DOI: 10.2514/6.2017-0419. URL: http://dx.doi.org/10.2514/6.2017-0419.

[37] Hazem Torfah. "Stream-Based Monitors for Real-Time Properties". In: *Runtime Verification*. Springer International Publishing, 2019, pp. 91–110. ISBN: 97830303207-99. DOI: 10.1007/978-3-030-32079-9_6. URL: http://dx.doi.org/10.1007/978-3-030-32079-9_6.

[38] Falcone Ylies, Havelund Klaus, and Reger Giles. "A Tutorial on Runtime Verification". In: *Engineering Dependable Software Systems*. IOS Press, 2013. DOI: 10.3233/978-1-61499-207-3-141. URL: http://dx.doi.org/10.3233/978-1-61499-207-3-141.