Tracing Correctness: A Practical Approach to Traceable Runtime Monitoring

Saarland University

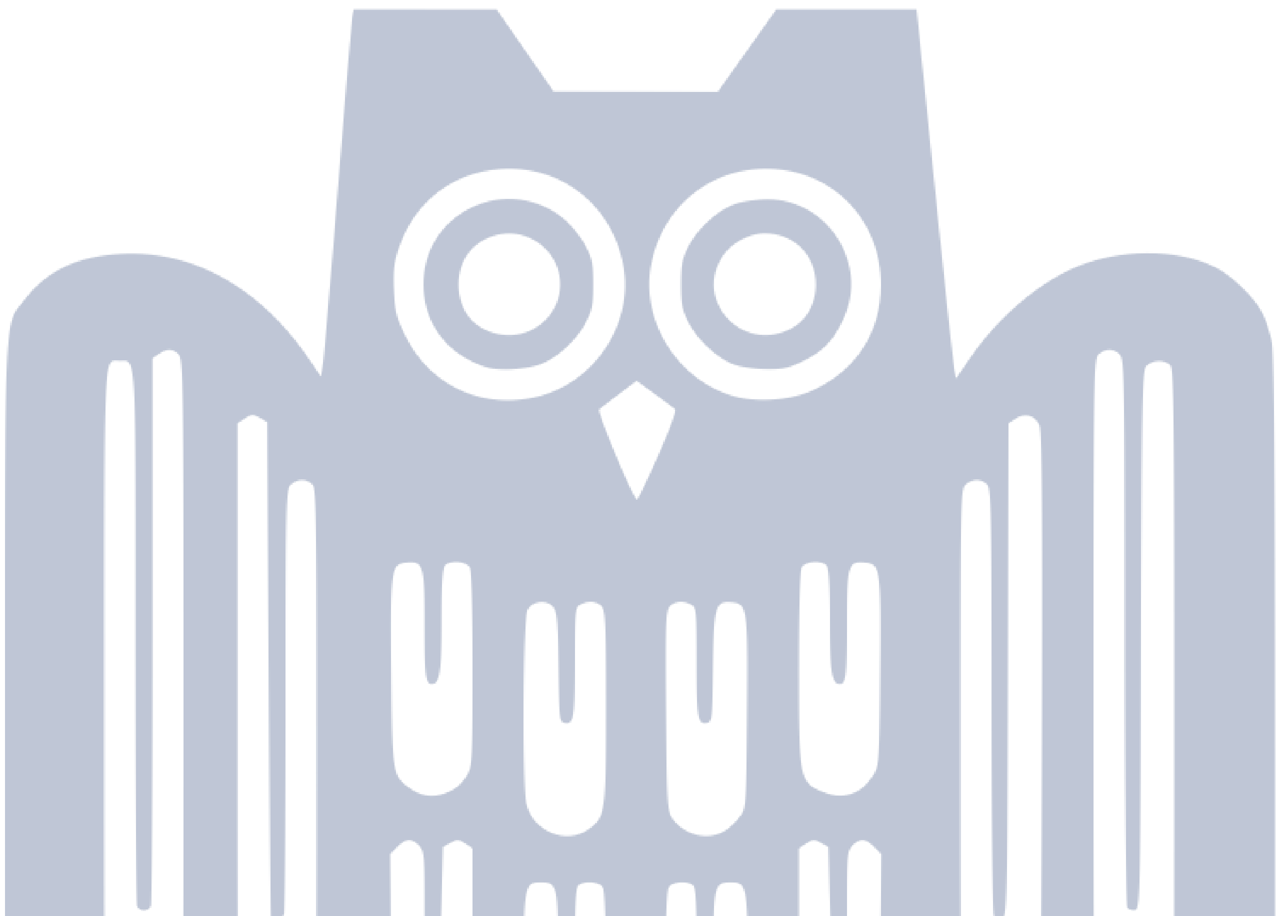
Department of Computer Science

Master's Thesis

submitted by

Jan Eric Baumeister

Saarbrücken, February 2020



Supervisor:	Prof. Bernd Finkbeiner, Ph.D.
Advisor:	Maximilian Schwenger
	Sebastian Schirmer
Reviewer:	Prof. Bernd Finkbeiner, Ph.D.
	Prof. Dr.-Ing. Stefan Levedag
Submission:	10 February, 2020

Abstract

The established approach to check the correctness of a cyber-physical system is to verify all possible executions statically. With their increasing autonomy, verifying all traces develops into a harder problem. To handle the increased complexity, runtime monitoring verifies the system dynamically.

RTLola is a stream-based specification language to express complex real-time constraints that provides different static analyses as the determination of an upper bound on the required memory. The underlying monitoring framework, StreamLab, uses these analyses and interprets the given RTLola specification. However, to find an industrial application for aerial vehicles, a certification by the Federal Aviation Administration (FAA) or the European Union Aviation Safety Agency (EASA) is needed. Part of this certification process is to show that the product is traceable, i.e., describing the relationship between the specification language and the software. One advantage of a traceable implementation is the identification and documentation of each code fragment and reason for their existence. For an interpreted monitor, like the StreamLab framework, it is unfeasible to show this property.

Recent work introduces a hardware-based approach, compiling `VHDL` code out of an RTLola specification. This `VHDL` code can then be synthesized into an FPGA implementation. This thesis presents a prototype implementation of this compilation concerning a traceable result. It additionally describes the integration of a synthesized monitor into the unmanned aerial vehicle superARTIS by the German Aerospace Center (DLR).

Acknowledgements

First, I wish to thank my supervisor Bernd Finkbeiner for giving me student research assistant jobs, which lead to this thesis. Moreover, I would like to thank the Unmanned System Department at the German Aerospace Center (DLR) Institute of Flight Systems in Brunswick for giving me the chance to write this thesis based on our cooperation. Further, I thank my advisor Maximilian Schwenger not only for his support during the thesis but also for advising my work as a student research assistant. He gave me excellent guidance and constructive feedback over the last two years. I am especially thankful that he always pushed me to become better in writing and never giving up. Another grateful thanks to my advisor Sebastian Schirmer for his enthusiasm towards monitoring. He always found time to support me. Thanks to him, my time in Brunswick was very exciting with a lot of work, but always with a smile on my face. Moreover, I would like to thank all DLR employees for their support, especially Nikolaus Ammann and Christoph Torens. Furthermore, I want to thank Prof. Finkbeiner and Prof. Levedag, for reviewing this thesis. Last but not least, I would like to thank my family and friends for their ongoing support, with special thanks to Tom Baumeister, Niklas Metzger, and Christian Schön for proofreading this thesis.

Eidesstattliche Erklärung

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

Statement in Lieu of an Oath

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

Einverständniserklärung

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

Declaration of Consent

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

Saarbrücken, 10 February, 2020

Contents

1. Introduction 1

2. Related Work 5

3. Background 7
   - 3.1. The specification language RTLola . . . 7
     - 3.1.1. Concrete Syntax & Abstract Syntax Tree . . . 7
     - 3.1.2. Dependency Graph . . . 10
     - 3.1.3. Type System . . . 17
     - 3.1.4. Finite Memory Monitoring . . . 24
   - 3.2. Hardware Compilation . . . 32
     - 3.2.1. General Structure . . . 33
     - 3.2.2. Notation . . . 34
     - 3.2.3. High-level Controller . . . 35
     - 3.2.4. Low-level Controller . . . 41
   - 3.3. VHDL & FPGA . . . 48
     - 3.3.1. VHDL Example . . . 48
     - 3.3.2. FPGA . . . 51

4. Prototype 53
   - 4.1. Entity Structure . . . 55
   - 4.2. Realization of Stream Types . . . 56
   - 4.3. High-Level Controller . . . 58
     - 4.3.1. TimeSelect . . . 58
     - 4.3.2. The Scheduler Template . . . 59
     - 4.3.3. CheckNewInput . . . 61
     - 4.3.4. ExtInterface . . . 61
     - 4.3.5. EventDelay . . . 62
     - 4.3.6. HLQInterface . . . 64
     - 4.3.7. High-level Controller . . . 66

4.4. Low-Level Controller . . . 67
4.4.1. Input Streams . . . 67
4.4.2. Output Streams . . . 69
4.4.3. Sliding Windows . . . 72
4.4.4. Evaluator . . . 78
4.4.5. Low-level Controller . . . 82
4.5. Monitor . . . 83

5. Integration 85
5.1. The Mission . . . 85
5.2. Integration Setup . . . 87
5.3. BoardSetup . . . 89

6. Case Study 93
6.1. RTLola Specifications . . . 93
6.1.1. Geofencing . . . 93
6.1.2. Sensor Validation . . . 97
6.1.3. Cross Validation in RTLola . . . 101
6.2. Evaluation . . . 106
6.2.1. Static Analysis . . . 106
6.2.2. Validation . . . 111

7. Conclusion & Future Work 113

A. Appendix 115
A.1. Inference Rules Type System . . . 115
A.2. Templates . . . 116
A.2.1. TimeSelect . . . 116
A.2.2. CheckNewInput . . . 117
A.2.3. ExtInterface . . . 117
A.2.4. HLC . . . 118
A.2.5. LLQInterface . . . 118
A.3. Realization of Stream Expressions . . . 119
A.3.1. Sliding Window Realizations . . . 125
A.4. Roadmap to Integrate the monitor in Vivado . . . 128
A.5. Geofence . . . 133
A.6. Static Analyzes . . . 136
A.6.1. Cross Validation . . . 136
A.6.2. Sensor Validation . . . 138

Chapter 1

Introduction

There are several ideas to use unmanned aircraft vehicles (UAV) for industrial applications, e.g., transport and reconnaissance, where these systems should fly autonomously. Due to the safety-critical aspect of aircraft, unexpected behavior of the autonomous system such as changing the computed flight path can result in a catastrophe. Therefore aviation companies have to ensure that their autonomous systems do not violate any critical constraints.

One approach to check the correctness of cyber-physical systems, i.e., systems that interact with the environment, is testing. Testing does not cover every possible behavior of the system, such that the absence of bad behavior cannot be guaranteed. Thus in safety-critical areas, stronger methods are needed. This goal can be reached with static verification methods like model checking. Model checking requires a model of the system and analyzes all possible runs in this model against a defined property afterward. If every path satisfies the specification, the absence of bad behavior, covered by the model and the specification, during the execution of the system is also verified. However, with increasing autonomy, the complexity of these systems increases dramatically. Therefore the size of the model also increases, such that these approaches might not be applicable.

In this thesis, we solve this issue with runtime monitoring [1, 2, 3, 4], a dynamic verification technique. In runtime monitoring, the behavior of the system is specified similar to model checking, but only the current run is analyzed, which reduces the complexity of the verification process. The idea of runtime monitoring is not to prevent bad behavior but to detect it and trigger an alarm to initiate countermeasures. This method can be applied either online or offline. In online monitoring, a separate monitoring component runs during the execution of the monitored system and analyses incoming data. In offline monitoring, the system produces a log file, which the monitor analyzes afterward.

To obtain formal guarantees on the behavior of the system, we need a formal specification language, e.g. temporal logic [5, 6, 7, 8]. A temporal logic consists of atomic propositions to express the current system state in an abstract way as well as boolean

and temporal connectives. Another type of specification languages are stream-based specification languages [9, 10, 11]. In comparison to temporal logics, such specifications are written data orientated like programming languages instead of logic orientated. In this thesis, we use the stream-based specification language RTLola [12], which is capable of expressing complex real-time constraints. Additionally, the monitor also provides formal guarantees and analyses, e.g. an upper bound on the memory consumption, as opposed to programming languages.

Incoming sensor data is defined as input streams in RTLola. This input data can then be combined into output streams to get statistical information. The specification language differentiates between event-driven output streams and periodic output streams, which differ in the so-called activation condition, i.e. the mechanism that initiates the update of a stream. As an example, to compute the average velocity during a complete run of an aircraft, the specifier defines an event-driven output stream that updates its value when the monitor receives a new velocity value. However, to compute the average velocity every 10s, a periodic-stream is needed. This stream evaluates its value with a frequency of 0.1Hz independent of the number of received new values. To express a violation, RTLola uses triggers that consist of a boolean expression and a message that raises the alarm when appropriate. The underlying framework StreamLab [13] receives an RTLola specification, analyses it and interprets the received input values based on the specification.

The current implementation of StreamLab is not applicable to safety-critical domains in the industry. As an example, we again consider aviation. To integrate hardware or software into aircraft, a certification process has to be passed. Part of this certification process is to connect the abstract specification with the concrete realization. For this connection, code fragments in the realization have to be annotated to relate them with their corresponding parts in the specification. However, the current StreamLab backend is an interpreter and not a compiler. In an interpreter, the input specification is encoded as data. Such data fragments are generated dynamically and cannot be annotated statically, which is demanded by the certification process. To solve this issue, we have to consider the specification as arbitrary, but fix in the annotation process. However, connecting the specification and the realization with such a representation is not as distinct as considering a specific specification. This challenges the annotation process and would be too costly in practice. In comparison, a compilation takes as input solely a specification and compiles a realization that can only monitor the incoming specification. This compiled monitor contains for each stream code fragments instead of data fragments. With these input specific code fragments, the relationship between a concrete specification and its corresponding code fragments can now be described.

Recent work [14] introduces a compilation from an RTLola specification to the Very High-Speed Integrated Circuit Hardware Description Language (VHDL) [15], which can then be synthesized into a Field Programmable Gate Array (FPGA). One advantage of a hardware-based solution is the parallel computation of different stream values without producing an overhead in comparison to a software-based solution. Another

important criterion for using hardware is the reduced power consumption that is limited in embedded systems.

This thesis describes a prototype implementation based on the theory for a hardware-based monitor, compiling an RTLola specification to VHDL code. Additionally, the compilation identifies and documents the code fragments concerning the specification, moving forward in the certification process. To test the real-world application of the monitor, we integrate our prototype into the Autonomous Rotorcraft Testbed for Intelligent Systems (ARTIS)[1] to monitor UAVs. This system is used for the development and evaluation of components for an autonomous flight system from the German Aerospace Center (DLR). It consists of a software framework to simulate the flight of a UAV as well as a fleet of unmanned aircraft. First, we integrated the monitor in the hardware-in-the-loop flight simulation that replays a flight based on log files. Next, we integrate the monitor on the superARTIS UAV to monitor the system during an actual flight. This integration results in a case study of monitoring UAVs with the specification language RTLola using FPGAs.

[1] https://www.dlr.de/ft/en/desktopdefault.aspx/tabid-1387/1915_read-15851/

Chapter 2

Related Work

To use formal runtime verification, we need a formal specification language. One approach is temporal logic [5, 6] consisting of atomic propositions, boolean arithmetic, and temporal operators. The constructed monitoring tools reach from inline methods to outline methods. The first approach adds assertions in the monitored system based on the specification. The second one constructs the monitor as a separate component out of the specification. One of these approaches is P2V [16] that compiles assertions written in the specification language sPSL [17] to synthesizable Verilog code. Another construction for a hardware-based monitor with temporal logic is introduced by Finkbeiner and Kuhtz [18], to monitor LTL specifications that can have unbounded future constraints with an FPGA implementation. The development of real-time temporal logics such as metric temporal logic (MTL) [7] and signal temporal logic (STL) [8] affected the engineering of new monitoring tools, e.g. Jaksic et al. [8] introduce an FPGA monitor that can express real-time constraints with STL. These new logics have the advantage of expressing timing behavior with real-time values instead of discrete ones. However, one drawback when monitoring systems with temporal logics is their expressiveness. With temporal logics, it is only possible to express the satisfaction or the validation of a property which is often not sufficient when monitoring cyber-physical systems, e.g. the time of the invalidation may not be the time of the system failure. To cover this problem, Moosbrugger et al. [19, 20] introduced the monitoring tool R2U2, which uses not only MTL to detect a validation of a property but also Bayesian networks to reason about the failure.

Another approach to increase the expressiveness but still have formal guarantees are stream-based specification languages. In stream-based specification languages, the user declares input streams, representing the input data, and output streams that contain calculations like arithmetic expressions but also future or past offsets to access future and past values. Because stream-based specification languages like Lola [10], Lustre [9, 21] or Copilot [11] were initially developed to monitor digital circuits, they assume that all input data is received synchronously, i.e., each input stream receives a new

value at the time, which is the case in circuits. However, this assumption does not hold when monitoring cyber-physical systems in general and restricts the expressiveness of the described properties. New approaches like RTLola [12], based on Lola, covers this problem by receiving input events with a non-fix rate, e.g. inputs are received without a fixed frequency. RTLola also introduces real-time features to Lola, like adding a frequency to output streams, describing the period when the value of an output stream should be computed, or the aggregation of streams over a specified period. Another extension of Lola to handle asynchronous event streams and to describe real-time properties is TeSSLa [22]. TeSSLa is a monitoring tool that reconstructs the program flow of a C program by receiving events from the embedded trace unit of the microprocessor and monitors this reconstruction. In comparison to RTLola, this requires information about the program flow of the monitored software when writing the specification. Besides, TeSSLa cannot aggregate over a stream for a specified period. A further extension of Lola is Striver [23]. Striver handles asynchronicity of streams by annotations, describing the period when a value is updated. In comparison to RTLola, they do not provide a mechanism to access a value from a fixed-rate stream in streams that have a variable-rate. RTLola realizes this by sliding windows and the sample-and-hold operator.

Adolf et al. [24] integrated Lola into an unmanned aerial vehicle (UAV) similarly to our approach. They described real-time constraints with discrete time stamps and the current system time, e.g. the property that the frequency of the incoming data is in the given bound. The main difference to our approach is the new specification language RTLola. RTLola enables us to express more complex real-time properties and to express the current ones in a more elegant way. Another difference is the resulting monitor, which is, in our case, hardware-based and not software-based. An integration of a hardware-based runtime monitor in a UAV is described by Moosbrugger et al. [19]. They integrated their runtime monitoring tool R2U2 synthesized onto an FPGA in the NASA DragonEye. As specifications, they introduce monitors to check the integrity of the system but also specifications to recognize hacking attacks at the UAV. The main difference to our approach is that the R2U2 tool uses temporal logic instead of a stream-based specification language.

Chapter 3

Background

This chapter describes the general structure of the specification language RTLola, its hardware-based realization, and introduces the hardware description language VHDL.

3.1. The specification language RTLola

In this thesis, we use RTLola [12] to describe properties in a formal way. RTLola introduces asynchronous monitoring and real-time constraints to the stream-based specification language Lola [10]. The following section first describes the concrete syntax of "vanilla" RTLola, the specification RTLola without any syntactical sugar such as infix notation for arithmetical operators. Then, we describe the transformation from the concrete syntax to an abstract syntax tree (AST). From this AST, we perform static analysis, i.e. a type checking analyzes. Afterward, we take a closer look at the semantics of the specification language.

All definitions, lemmas, and propositions are introduced and proven by Schwenger [25].

3.1.1. Concrete Syntax & Abstract Syntax Tree

The formal semantics of RTLola is defined on the abstract representation of specification. For this reason, the function *AST* transforms the concrete syntax of a specification to an abstract syntax tree (AST), with $s_n^{\downarrow} + s_n^{\uparrow} + s_n^{!}$ children. Note that from this point on, we uses marker to indicate the category of a stream. Input streams are symbolized with a down arrow ($s_s^{\downarrow}$), output streams with an up arrow ($s_s^{\uparrow}$), and trigger with an exclamation mark ($s_s^{!}$). To indicate that a stream can be either an input stream or output stream, we use a vertical line (s_s^{-}). The following paragraphs describe the concrete and abstract syntax for input streams, output streams, and triggers.

The ith input stream $s_i^{\downarrow}$ is defined by:

```
input a: T
```

The output of the *AST* function for input streams is $s_i^{\downarrow} = (a, AST(T))$. The name a of the stream is irrelevant for the formal semantics, but needed in the realization. However, the type T needs to be transformed into an AST object. For readability, we define $s_i^{\downarrow}.name := a$ and $T_i^{\downarrow} := AST(T)$.

The jth output stream $s_j^{\uparrow}$ is defined by

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

This syntax is similar to input streams, but introduces two more components: the evaluation frequency @nHz and the expression expr. The AST representation for output streams is $s_s^{\uparrow} = (a, AST(T), n, AST(expr))$, and respectively $s_j^{\uparrow}.name := a$, $T_j^{\uparrow} := AST(T)$, $s_j^{\uparrow}.ext := n$, and $s_j^{\uparrow}.expr := AST(e)$. We restrict the frequency to a positive natural number n with unit Hertz. The resulting monitor and the specification language is theoretically capable of working with other positive rational numbers and other time units. However, for simplicity reasons in the semantics, we perform theses restrictions.

The specified frequency in output streams is an optional value. The following concrete syntax for an output stream with the AST representation $s_s^{\uparrow} = (a, AST(T), \bot, AST(e))$ is also possible:

```
output a: T := expr
```

The kth trigger $s_k^{!}$ is defined by:

```
trigger a "msg"
```

Trigger contain a stream name a and a message msg. The *AST* transforms the name to the corresponding stream reference if the name is specified and to $\bot$ otherwise. This transformation results in the following AST representation, with $s_k^{!}.tar := s$ and $s_k^{!}.msg := msg$:

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

Expressions

RTLola supports different computation rules for output streams. For each expression, we define the concrete syntax and the abstract representation as output of the *AST* function, starting with different accesses:

Synchronous Lookup: The expression s accesses the current stream value of s and bounds the timing of the accessor stream to the schedule of the accessed stream. The abstract representation for this expression stores a reference to the accessed stream:

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

Sample & Hold Lookup: The concrete syntax for the sample & hold lookups is defined by: `s.hold()`. Sample & hold lookups access the current stream value of `s` as synchronous lookups. However, these lookups are considered asynchronous, i.e. the timing of the accessor stream is not bounded to the schedule of the accessed stream. The abstract representation for this expression also stores a reference to the accessed stream, resulting in:

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

Offset Lookup: Offset expressions `e.offset(by: -n)` correspond to synchronous lookup accessing not the current, but the nth previous value. Like synchronous lookups, the timing of the accessor stream is also bounded to the schedule of the accessed stream. This expression results in the following abstract representation storing the offset number n besides the stream reference and the access category:

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

Sliding Window Lookup: Sliding windows `s.aggregate(over: δs, using: γ)` aggregate over all values inside a real-time duration δ with an aggregation function γ. These timed accesses are asynchronous and do not influence the timing behavior of the accessor stream, as Sample & Hold lookups. The abstract representation contains therefore the stream reference, the time duration and the aggregation function:

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

Default Expression: Because some lookups can fail, the return type of sample & hold, offset, and sliding window lookups are optional types. RTLola uses default expressions `e.defaults(to: e)` returning the default value in case of a lookup fail. This results in the abstract interpretation:

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

Function Expression: Function expressions `f(e1,...,en)` are used to refine the received input data. RTLola supports the usual operators with the infix notation, such as constants, arithmetic operations, or conditionals encoded as n-ary functions. Note that, because we encode the common condition `if cond then cons else alt` as a ternary function, we evaluate this expression without side effects. Additionally,

RTLola implements the functions for the absolute value or the square root. The abstract representation is given by:

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

Note that for sliding windows as well as for streams, we assign each sliding window to a unique identifier. Therefore, the lth sliding window in the specification is identified by w_l.

The first constraint for a valid RTLola specification is the syntactical validity:

Definition 1 (Syntactic Validity [25])

Def. Syntactic Validity

An RTLola specification is *syntactically valid* iff

- all stream and trigger definitions conform to the concrete syntax stated above.
- $AST(s)$ can be computed without violating a condition. This especially includes that all stream names can be resolved, such that no $\bot$ value is contained in the AST.
- all names of streams are unique, i.e., $\forall i, j : s_i^{-}.name = s_j^{-}.name \implies i = j$.

Before we start with the constraints for semantical validity, we introduce some further notation. We denote $n^{\downarrow}$ as the number of input streams, $n^{\uparrow}$ the number of output streams, $n^{!}$ the number of trigger, and n^{w} the number of sliding windows. Additionally, we call the set of all input streams in a specification $Stream^{\downarrow}$, the set of all output streams $Stream^{\uparrow}$, the set of all triggers $Stream^{!}$, and the set of all streams $Stream$. The set $\mathcal{W}$ defines the set with all sliding windows. These sets are defined with:

$$Stream^{\downarrow} = \{s_i^{\downarrow} \mid i \leq n^{\downarrow}\}$$

$$Stream^{\uparrow} = \{s_i^{\uparrow} \mid i \leq n^{\uparrow}\}$$

$$Stream^{!} = \{s_i^{!} \mid i \leq n^{!}\}$$

$$Stream = Stream^{\downarrow} \dot{\cup} Stream^{\uparrow} \dot{\cup} Stream^{!}$$

$$\mathcal{W} = \{w_i \mid i \leq n^{w}\}$$

In this chapter, we refer from this point in time to the AST instead of the concrete syntax.

3.1.2. Dependency Graph

The previous section defined the syntactical validity of a specification. Before we can build a monitor out of the specification, we need to define the semantical validity. Section 3.1.1 introduced different stream accesses occurring in the stream expression. RTLola uses the definition of a dependency graph (DG) to describe the dependencies

between streams. Then, the DG is used for semantic checks like the type analysis or the existence of an evaluation model. Additionally, we determine from the DG a static memory upper bound for each stream. These upper bounds are used from the compiler to determine the number of registers in the monitor realization.

RTLola builds the DG from the AST by iteration over it. We describe a dependency as a triple consisting of the source stream, the weight, and the target stream. The weight of a dependency is either the offset which value needs to be addressed, or a pair with the duration and aggregation function for sliding windows. The dependencies for an output stream $s_i^{\uparrow}$ is computed recursively from the stream expression. With these dependencies, RTLola builds the dependency graph.

Definition 2 (Dependency Graph [25])

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

Def. Dependency Graph

$$V := Stream$$

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

The function $dep_{s_i^{\uparrow}}$ is defined as:

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

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

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

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

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

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

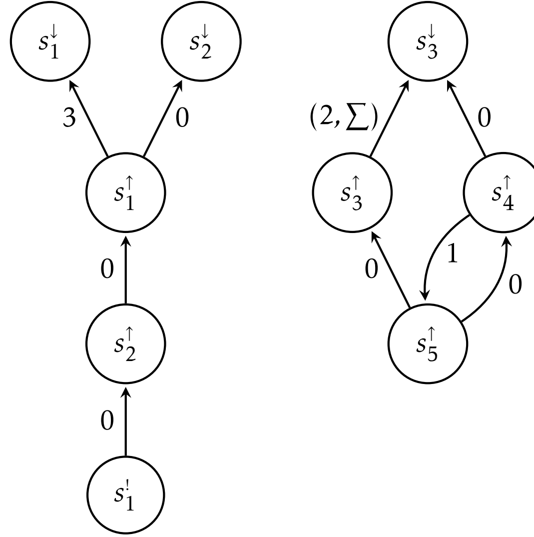


Figure 3.1.: Dependency Graph for the specification in Example 3.1.1.

Example 3.1.1. Figure 3.1 represents the dependency graph for the following specification:

```
input a : Float32
input b : Float32
input c : Int64
output d : Float32 := a.offset(by: -3).defaults(to: 0.0) + b
output e : Bool := d < 40.5
output f : Int64 @1Hz := c.aggregate(over: 4s, using: ∑)
output g : Int64 := c * h.offset(by: -1).defaults(to: -4)
output h : Int64 := f.hold().defaults(to: 50) - g
trigger e "e is smaller than 40.5"
```

Input streams do not access other stream and are always sink nodes. Whereas trigger are always leaves nodes, because they cannot be accessed by other streams. The dependencies of output streams is defined by their lookups in the stream expression. The output stream d for example has an offset lookup to the input stream a and a synchronous lookup to the input stream b. Therefore, the dependency graph has a 3-weighted edge from $s_1^\uparrow$ to $s_1^\downarrow$, and a 0-weighted edge from $s_1^\uparrow$ to $s_2^\downarrow$. △

Evaluation Models

With the DG, we start with the first semantical analysis for a RTLola specification without knowing the concrete semantics. RTLola uses an evaluation model to describe the relation between input and output streams in the semantics. Because the formal definition of an evaluation model is not needed to check the semantical validity, we just give an intuitive definition: An *evaluation model* of a specification is a set of infinite sequences, one for each stream. To get an intuition for an evaluation model, consider the following specification:

Evaluation Model

```
input a : Int8
output b : Int8 := a + 1
```

The specification evaluates the output stream b to the value from input a plus 1. Intuitively, the unique evaluation model for this specification follows the specification and adds one to the input value. Formally, this is described by:

$$\{(s_a^{\downarrow}, x), (s_b^{\uparrow}, y) \mid x, y, z \in \mathbb{N}^{\omega} \wedge x_i = (z_i \; mod \; 128) \wedge y_i = (z_i + 1 \; mod \; 128)\}$$

However, there are specifications which have more than one valid evaluation model like:

```
output a : Int8 := a
```

This specification consists only of one output stream, which assigns the output stream a to the value of a. This specification is syntactically valid, but we can find different evaluation models. One possible model is similar to the first part of the previous example. The model assigns a to each value inside the modulo class. Another model, which assigns a to a constant value c, is also possible. Formally, we can describe these models as:

- $\{(s_a^{\downarrow}, x) \mid x, z \in \mathbb{N}^{\omega} \wedge x_i = (z_i \; mod \; 128)\}$
- $\{(s_a^{\downarrow}, x) \mid x \in \mathbb{N}^{\omega} \wedge c \in \mathbb{N} \wedge x_i = c\}$

Another possibility for a syntactically valid specification is the absence of an evaluation model. Consider the following specification:

```
output a : Int8 := b
output b : Int8 := a + 1
```

In this example, the value of a is assigned to the value of b, and b to a plus 1. Because of the circular dependency, we cannot find two sequences fulfilling the specification. To guarantee that the monitor provides exactly one evaluation model such that for the same input sequences the monitor computes the same output sequences, the definition of well-definedness was introduced for Lola.

Definition 3 (Well-definedness [10])
A specification is said to be *well-defined* if a unique model exists.

Def. Well Definedness

To check the existence of a unique evaluation model for a specification, Lola checks the specification for well-formedness:

Definition 4 (Well-formedness [10])
A Lola specification is *well-formed* if there is no 0-weighted cycle in the dependency graph.

Def. Well-formedness

The proof of the following theorem confirms that well-formedness is a sufficient but no necessary criterium.

Theorem 1 (Unique Evaluation Model [10])**.** *A specification has a unique evaluation model if the specification is well-formed. The inverse direction does not hold.*

The intuitive explanation for the correctness of this theorem results from an order of the stream, which can be determined with the absence of a 0-cycle. This order describes the schedule of the stream evaluation and is described in the next section more detailed. An example for the inverse is presented by the following stream:

```
output a : Bool := a ∨ ¬a
```

The unique evaluation model for this stream is a sequence which always returns `true`. However, the dependency graph has a 0-edge from the stream `a` to itself, resulting in a 0-weighted cycle.

To adapt this theorem to RTLola, we need to consider the asynchronous lookups sample & hold and sliding window. In both cases, the expression requires access to the target stream. We first look at their representations in the DG. Sample & hold lookups are represented as synchronous accesses, with a 0-weight edge. The difference between these lookups is the timing of their evaluation. However, this is irrelevant for the dependency description, so they are treated in the same way. Sliding windows are represented by the duration of the window and the aggregation function. The computation of the sliding window requires an iterative lookup on the current value of the target stream. For the evaluation model, the number of accesses is irrelevant as long as only the current value is accessed. So we treat sliding window lookups as 0-weight edges. With these adaptions, RTLola uses the same definitions for well-formedness. The proof for this adaption and the resulting theorem is presented by Schwenger [25].

Example 3.1.2 (Well-formedness)**.** Consider again the dependency graph from Figure 3.1. This graph contains only one cycle between $s_4^{\uparrow}$ and $s_5^{\uparrow}$, which a weight of one. So the dependency graph has no 0-weighted cycle, and the specification is well-formed. △

Evaluation Order

To prove the existence of a unique evaluation model, RTLola computes an evaluation order from the DG. This order describes the schedule of the stream evaluations. To motivate the defined evaluation order and to get an intuition, we discuss at first two examples.

Example 3.1.3. The following specification declares one input stream `a` and two output streams `b` and `c`. During the evaluation `b` adds up the constant number `3` to the input value. The output stream `c` then adds up the current value of `b` to the previous one.

```
input a : Int8
output b : Int8 := a + 3
output c : Int8 := b + b.offset(by:-1).defaults(to: 0)
```

The input stream `a` is synchronously accessed by the output stream `b`, such that `a` needs to be evaluated before `b`. The output stream `c` accesses the current value and the previous one of `b` synchronously. For this reason, the output stream `c` evaluates the stream expression after `b`. These accesses result intuitively in the order $a \prec b \prec c$. △

Example 3.1.4. Consider the following specification:

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

Compared to the previous example, the specification adapted the stream expression of output stream b to add up a with the previous value of c. Using the same approach as in the previous example results in the following problem. Because b accesses c and c is accessed by b, we cannot define a total order in the same way. Because b accesses the previous value, which is already available without a computation of c, b can theoretically compute the stream expression before c. However, if we evaluate b before c a new problem arises. Since c was not updated yet, the stream offset lookup in b accesses the second last value and not the previous one as intended. △

This issue is solved with a new evaluation phase — called *pseudo evaluation phase*. This phase assigns the current value of each output stream, which will be evaluated, to a pseudo value. With this assignment, we shift the offset values to their correct position such that the stream expressions using the offset values can compute their stream expressions. In our example, with this shift, we can evaluate b after the synchronous direct access stream a was evaluated, and c after b. RTLola replaces the pseudo value with the correct one, when the stream is evaluated based on the evaluation order.

Pseudo Evaluation Phase

RTLola uses the dependency graph to receive the evaluation order for a specification. Because the pseudo evaluation phase covers the offset lookups such that offset values are at their intended position, the computation only considers 0-weight edges and sliding windows. Intuitively, if a stream $s_i^{\uparrow}$ has a direct access to stream s_j^{-}, it needs to be evaluated before $s_i^{\uparrow}$. This intuition results in the following definition:

Definition 5 (Evaluation Order [25])

The *evaluation order* $\prec$ is a partial order on streams, reflecting the structure of the dependency graph $DG = (V, E)$. The evaluation order is the transitive closure of a relation satisfying the following rules:

Def. Evaluation Order

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

Intuitively, the first rule restricts the evaluation of input streams before the evaluation of output streams. This is in general not necessary but simplifies the evaluation. The last rule covers the previously mentioned dependencies between two streams.

Remark 3.1.1. *The definition from Schwenger [25] contains three rules. In the previous definition, the rules correspond to the first and the third rule. In this thesis, we omit the second rule, covering the transitivity of the order. Because the definition already claims that the evaluation order is a partial order, this implicitly includes the constraints reflexivity, transitivity, and antisymmetry, such that the second rule is already covered.*

The order between two streams, which are incomparable in the evaluation order, can be evaluated independently of each other, i.e. the order between those streams does not have an influence on the stream evaluation. These streams either do not depend on each other or all needed values are already accessible. Because trigger values are not accessed by output streams, triggers are not part of the evaluation order and are computed after all output streams.

The following proposition and the corresponding proof guarantees the existence of an evaluation order for well-formed specifications.

Proposition 2 (Existence of Evaluation Order [25]). *Every well-formed specification has an evaluation order.*

The proof first constructs a relation out of the dependency graph with the constraints from Definition 5 and then proves that the reflexive transitive closure is a partial order.

For readability, we introduce a new representation for the evaluation order.

Definition 6 (Evaluation Layer [25])

Def. Evaluation Layer

The *evaluation layer* is an equivalent representation of $\preceq$. If $Layer(s_i^-) = k$ then there is a strictly decreasing sequence of k streams w.r.t. $\prec$ starting in s_i^-.

For readability, λ^{max} describe the maximum layer of a specification, formally defined as:

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

→ Ex. 3.1.1, P. 11

Example 3.1.5 (Evaluation Layer). The specification from Example 3.1.1 results in the following evaluation layers:

$$Layer(s_1^\downarrow) = 0 \qquad Layer(s_2^\uparrow) = 2$$

$$Layer(s_2^\downarrow) = 0 \qquad Layer(s_3^\uparrow) = 1$$

$$Layer(s_3^\downarrow) = 0 \qquad Layer(s_4^\uparrow) = 1$$

$$Layer(s_1^\uparrow) = 1 \qquad Layer(s_5^\uparrow) = 2$$

Input streams are always in layer zero. The output streams `d` and `f` are on the first layer, because they depend only on the input streams. The output stream `e` has a synchronous lookup to the output stream `d`. For this reason, its layer is after the layer of `e`. The output stream `g` has a synchronous lookup to the input stream `c` and an offset lookup to `g`, which is irrelevant for the evaluation order. Therefore, its layer is the same layer as the layer of `d` and `f`. The layer of the output stream `g` is greater than the layer of `f` and `g`, because of the synchronous lookup and the sample & hold lookup. △

3.1.3. Type System

A type checking analysis is part of the semantical validity, finding inconsistencies in the stream evaluation. To find these inconsistencies, the concrete syntax demands that each stream is annotated with the corresponding type. However, most of the output stream types can be inferred from the stream expression, such that the realization does not have this restriction.

In RTLola, a type is a pair consisting of the *value type* and the *activation condition*. The value type defines the bit range and its interpretation of the stream values, i.e. the input and output values of the monitor. Possible representations are booleans, signed and unsigned integers, as well as floating point numbers. This enumeration results in the following definition:

Definition 7 (Value Types [25])

Def. Value Type

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

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

Remark 3.1.2. *In the original definition Float(16) is not mentioned. Because a specification is realized on hardware in this thesis, space is highly limited. Therefore, we introduce an additional floating type, with a smaller bit representation.*

The activation condition of a stream defines the timing behavior, i.e. the condition when a stream needs to be updated. RTLola separates output streams into two categories, event-based and periodic, reflected in the activation condition. Periodic streams are annotated with a frequency, defining fixed time points, the stream is evaluated on. Event-based streams are streams without a frequency and therefore evaluate at arbitrary time points. The activation condition for theses streams is inferred from the stream expression. Therefore, we first collect all synchronous and offset lookups in the stream expressions. For this set, we then compute recursively all synchronous and offset lookups until we reach only a set of input streams. This resulting set then describes the activation condition. An output stream $s_i^{\uparrow}$ with activation condition ι is updated iff the incoming event updates *all* input streams $s_i^{\downarrow} \in \iota$.

Definition 8 (Activation Condition [25])

Def. Activation Condition

In RTLola, an activation condition σ is either an element of an event type $\iota \in ET$ or of a periodic type $\pi \in PT$. The set ET is defined as the powerset of all input streams in the specification, covering all possible input event combinations. The set PT is defined as the set of all time points where at least one periodic stream is evaluated. These sets are formally defined as:

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

$$PT := \left\{ p \mid p \in \mathbb{N} \wedge \left(\gcd(P^{spec}) \,\middle|\, p \right) \wedge \left(p \,\middle|\, \mathrm{lcm}(P^{spec}) \right) \right\}$$

The set P^{spec} is defined as the set of all periodic output streams:

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

Remark 3.1.3. *The previous definition derivates from the original one. In this thesis, we define single pacing type instead of defining the event type ET and periodic type PT, which does not change the intended definition. Additionally, we use a different notation. The activation condition is defined as a single value of the pacing type, which is the set of activation conditions of a specification.*

Remark 3.1.4 (Periodic Input Streams)**.** *In* RTLola*, the activation condition of an input stream is always event-based. To show the problem of a periodic input stream, we assume a specification containing the stream:* `input a : Int8 @1Hz`*. To evaluate the input stream* `a` *the monitor demands fixed time steps, exactly every second, for an incoming event updating* `a`*. However, in this case the specification has control of the input streams, which is in practice not possible. Even if input values are produced with a fixed frequency, the frequency cannot be guaranteed for the monitor because of internal delays and interferences. Therefore,* RTLola *excludes periodic input streams.*

Remark 3.1.5. *To annotate the activation condition ι of an event-based stream s^- in the specification, we use the following notation: s^- @ι. Note that ι is a set of input streams.*

To get an intuition for the activation condition, consider the next example.

Example 3.1.6 (Activation Condition)**.** The following specification presents a fragment of the specification in Example 3.1.1:

→ Ex. 3.1.1, P. 11

```
input a : Float32 @{a}
input b : Float32 @{b}
input c : Int64 @{c}
output d : Float32 @{a,b} := a.offset(by: -3).defaults(to: 0.0) + b
output e : Bool @{a,b} := d < 40.5
output f : Int64 @1Hz := c.aggregate(over: 4s, using: ∑)
```

The input streams `a`, `b`, and `c` are event-based. The activation condition of input streams is always an event-based type with the input stream itself because they depend on no other streams. The activation condition of the event-based output stream `d` is bounded to the input streams `a` and `b`, because of the synchronous lookup and the offset lookup. In this example, the output stream `d` is updated iff the monitor receives an incoming event which updates `a` and `b`. The event-based output stream `e` has a synchronous lookup to the output stream `d`. Therefore, it recursively depends on the input streams `a` and `b`, resulting in the same activation condition. The periodic stream `f` is annotated with a frequency and is, for this reason, updated every second, independent on the number of received events. △

From the AST, RTLola can assign each stream in the specification to a concrete type. However, to validate stream expressions we perform a type checking analysis. For this, we have to assign each expression to its corresponding type. Because expressions, in comparison to streams, are not annotated with their concrete types we have to infer them, resulting in a new problem. The following example visualizes this problem, assigning expressions to concrete types. Consider a constant expression 3, which can be a subexpression of s_i^- + 3. In case of $T_i^- = Int(8)$, the concrete type of 3 is $Int(8)$. However, if s_i^- has type $UInt(8)$, the concrete type of 3 is $UInt(8)$. For this, the type of an expression is not a single concrete type, but a candidate set of concrete types. The resulting candidate set for 3 is $\{Int(x), UInt(x) | x \in \{8, 16, 31, 64\}\}$. The candidate set of an expression is defined as the *abstract type*. Like the concrete type, the abstract type is defined as a pair, storing the value type and the activation condition.

Definition 9 (Abstract Types [25])

A single *abstract type* $(\widetilde{\tau}, \widetilde{\sigma})$ is defined as an element of $\widetilde{VT} \times (\widetilde{ET} \cup \widetilde{PT})$, with:

Def. Abstract Type

$$\widetilde{VT} := \{\emptyset, \{Bool\}\}$$
$$\cup \bigcup_{i=4}^{6} \{Float(y) \mid \{16, 32, 64\} \ni y \geq 2^i\}$$
$$\cup \bigcup_{i=3}^{6} \{\{UInt(y) | \{8, 16, 32, 64\} \ni y \geq 2^i\}, \{Int(y) | \{8, 16, 32, 64\} \ni y \geq 2^i\}\}$$
$$\widetilde{ET} := ET$$
$$\widetilde{PT} := PT$$

Intuitively, the abstract type is defined as:

- The single abstract value type $\widetilde{\tau}$ denoting the set of all possible values types.
- The single abstract periodic type $\widetilde{\pi}$ denoting the maximal frequency of how the stream can be evaluated.
- The single abstract event type $\widetilde{\iota}$ denoting the minimal set of dependencies of the expression.

From now on, we use the tilde to differentiate concrete and abstract types. Additionally, τ always references to the value type and σ the activation condition. Furthermore, π denotes a periodic type and ι an event type.

Type Lattice

RTLola uses a type checker which ignores the details about the underlying type system. With this approach the type system can be extended or modified without changing the

model checking algorithm. Nevertheless, to make constraints about the type system, we use the concept of a meet-semi lattice.

Definition 10 (Meet-semi Lattice)

Def. Meet-semi Lattice

A *meet-semi lattice* is a tuple $(S, \sqcap)$ consisting of a set S, ordered by a partial order $\preceq$, and a binary relation $\sqcap$, with the following constraints:

- Existence of a greatest lower bound between two elements: $\forall s_1, s_2 \in S. s_1 \sqcap s_2 \in S$
- Associativity of $\sqcap$
- Commutativity of $\sqcap$
- Reflexivity of $\sqcap$

The underlying meet semi-lattice for the current type system is defined as:

Definition 11 (Type Lattice [25])

Def. Type System in RTLola

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

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

The single meet operators are defined as:

- $\widetilde{\tau_1} \sqcap_{VT} \widetilde{\tau_2} := \widetilde{\tau_1} \cap \widetilde{\tau_2}$
- $\widetilde{\pi}_1 \sqcap_{PT} \widetilde{\pi}_2 := gcd(\widetilde{\pi}_1, \widetilde{\pi}_2)$
- $\widetilde{\iota_1} \sqcap_{ET} \widetilde{\iota_2} := \widetilde{\iota_1} \cup \widetilde{\iota_2}$

Intuitively, a meet operator usually refines two elements to a more concrete one. For this, the different meet operators have the following messages.

- An abstract value type contains all possible concrete value types. Therefore, the refinement of two abstract value types are all concrete types, which are contained on both sides. If the types are incomparable the meet operator returns $\emptyset$.

- An abstract periodic type is the maximal frequency on which the stream or expression can be evaluated. The result of the meet of two periodic types is the maximal frequency on which both streams can be evaluated, which is the greatest common divisor. This result is at least as slow as the input frequencies or even slower.

- An abstract event-based activation condition is a set containing all input streams, which need to be contained in an incoming event to evaluate the stream. Therefore, the set of the meet operator consists of all elements which are in one of the arguments, resulting in the union, with the bottom element $Stream^{\downarrow}$.

Remark 3.1.6 (Optional Types)**.** *Some expressions, i.e. offset lookups or sample & hold, return an optional value, because the monitor cannot guarantee the existence of a value. To see the need for optional return types in the lattice, we consider the following specification, assuming that optional values are treated as non-optional ones:*

```
input a : Int8
output b : Int8 := a + b.offset(by:-1)
```

This specification adds up all values from the input stream `a` *with the output stream* `b`*. Without using optional value types, it is intuitively clear, even without knowing the complete semantics of* RTLola*, that the type checker would allow the expression* `a + b.offset(by:-1)`*, because* `a` *and* `b` *have the same value type Int(8). However, in the first iteration the previous value of* `b` *is not defined resulting in an error. Therefore, to represent optional values in the type lattice, we introduce an additional abstract value type $Opt\langle\widetilde{VT}\rangle$, lifting the value types to optional types.*

Remark 3.1.7 (Meet-semi Lattice Proof)**.** *Schwenger [25] proved that the lattice defined in Definition 11 is a meet-semilattice, by proving associativity, commutativity, and idempotency for the meet operator $\sqcap$. The proof first shows the properties for each meet operator $\sqcap_{VT}$, $\sqcap_{ET}$, and $\sqcap_{PT}$ and then follows that the combination also results in a meet operator.*

Type Checking

The overall goal of the type checker is performing an analysis which finds expressions that are incompatible to each other. This includes expressions with different activation conditions, e.g. a synchronous lookup to an event-based stream from a periodic-output stream. But it also includes value type errors, e.g. multiplying a boolean stream with a number. Before we perform the type checking analysis, we define a relation $\models$ for type validity. We denote that an abstract type $(\widetilde{\sigma}, \widetilde{\tau})$ is a model for an expression e, with a binary relation $\models$ by: $\widetilde{\sigma}, \widetilde{\tau} \models e$. Based on this relation, the type checker uses inference rules, which are defined for each stream and expression individually. To apply the inference rules for expressions, we first need to lift the concrete stream value types to abstract ones. Therefore, the inference for streams uses the following generalization function:

Def. Generalization Function

Definition 12 (Generalization Function [25])
The *generalization function lift* : $VT \to \widetilde{VT}$ for a concrete value type τ is defined as:

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

In each inference rule we check for each stream and expression if the abstract value type and activation satisfies the expression or stream. If the expression or stream is a non-leaf node in the AST, we additionally check for each child if the abstract type satisfies the subexpression.

To get a intuition for the inference rules, consider the inference rule for default expressions:

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

A default expression returns a default value in case of a lookup fail, resulting in two subexpressions. For this, the inference rule checks the compatibility of the subexpressions: First, it finds the abstract value types and activation conditions for each subexpression. For these abstract types, the rule defines the following constraints: The first constraint checks if the abstract value type of the first subexpression is an optional value. Intuitively, to represent a lookup fail RTLola uses optional types. In case that the first expression does not return an optional type, i.e. no lookup fail is possible, the default value would never be taken and therefore the default expression is not needed. This behavior needs to be detected by the type checker. The second constraint then checks if the value type of both subexpressions are comparable, and refines the value type of the default expression. Because the default expression resolves the lookups fails, this check does not use the optional abstract type of the first sub expression, but the abstract value type stored in the optional one. The third constraint then compares and refines the activation condition.

The most restricted inference rule is for sliding window lookups, defined as:

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

In comparison to all other rules, sliding window lookups enforce abstract periodic types in the relation. Before, we give an intuition, we recap the behavior of sliding windows. A sliding window aggregates over all values of a stream s_i^- during a specified duration. For this reason, when reaching a time point, where to evaluate a stream expression $s_j^\uparrow$, with a sliding window lookup, the window needs to decide which values of s_i^- are part of the current window. If we restrict the activation condition of sliding windows

to a periodic type, the time points when evaluating a stream are statically known. As a consequence, the windows of the sliding window lookup are also statically known, which simplifies the decision which stream values are part of the window, even if they arrive in non-fixed time steps. Section 3.1.4 discusses the evaluation of sliding windows more detailed. Another constraint resulting from the sliding windows is the duration. Similar to the restrictions for the frequency, Schwenger [25] restricts the duration to a natural number simplifying the correctness proofs. However, in general RTLola is capable to describe constraints without these restrictions. Another constraint from the inference rule for sliding window lookups results from the aggregation function. To evaluate the window, the function uses the stream values from s_i^-. Therefore, the type $T_i^\uparrow$ needs to be compatible with the input type of the function. The same constraint holds for the return type, which needs to be compatible with the window lookup.

→Sec. 3.1, P. 24

Remark 3.1.8. *A list of all inference rules is presented in Appendix A.1. We do not discuss all rules separately. If you are interested in the details, a complete description of all inference rules is presented by Schwenger [25]. Nevertheless, we present in this thesis the general concept, with the following examples.*

→App. A.1, P. 115

With the inference rules and the binary relation, we define the *type validity* of a specification.

Definition 13 (Type Validity [25])

A specification has *valid types* if and only if for every stream and trigger there is a non-contradictory value type and activation condition.

Def. Type Validity

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

With type validity, we defined the last constraint for a valid specification, resulting in the definition:

Definition 14 (Specification Validity [25])

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

Def. Valid Specification

- *Syntactic validity* according to Definition 1 →Sec. 3.1, P. 10
- *Well-formedness* according to Definition 4 →Sec. 3.1, P. 13
- *Type validity* according to Definition 13 →Sec. 3.1, P. 23

Remark 3.1.9 (Formal Semantics)**.** *Schwenger [25] defined a formal semantics assuming infinite memory. Therefore, he first introduces the concept of relevant timestamps, which compromises all possible timestamps to the timestamps, where at least one activation condition of all streams is satisfied. Then, he defines the evaluation process with an infinite memory*

model, and afterward, the expression evaluation. The focus of this section is getting an intuition for the specification language, which provides a tradeoff between expressiveness and formal guarantees. If you need more details about the formal semantics of RTLola, *we refer to [25, 12].*

One essential guarantee for the hardware realization is a finite memory model. Therefore, if we analyze the dependency graph and restrict the aggregation function to get an upper bound for each stream entity and sliding window, we can compress the infinite model to a finite one. This analysis and the restriction on the aggregation functions are intuitively outlined in the following paragraph.

3.1.4. Finite Memory Monitoring

In the previous section, we defined the conditions for a valid specification. Before we describe a realization for such specifications, we first define memory upper bounds on the stream entities to realize the monitor with finite and statically known memory. This includes the evaluation of the activation type as well as for the expression evaluation. Additionally, we have to restrict the aggregation function for a finite memory realization.

Handling Time

→ Sec. 3.1, P. 17

Section 3.1.3 introduces two types of activation conditions, event-based and periodic. The activation condition of event-based streams is bounded to the incoming event, e.g. an output stream b with the activation condition $\{a, b\}$ is evaluated with each event that updates the input streams a and b. Because the monitor has no control about the timing of the incoming events, the timing of event-based streams cannot be determined statically. As a complement, period streams are annotated with a frequency which bounds the activation condition to fixed-time points, e.g. an output stream b with activation condition 2Hz is evaluated every 500ms. Due to the fixed time stamps, we can define statically a global *schedule* for all periodic streams. The entries of the schedule then contain a time stamp and a non-empty set of period output stream, with the following notation:

Schedule

Deadline

- We call a time stamp in the schedule a *deadline*.
- We say a deadline is due if the monitor reaches a timestamp which is part of the schedule.

Hyper-Period

To find a finite representation for the schedule, we use the hyper-period. The *hyper-period* Π of a specification is the least common multiple of all frequencies in the specification:

$$\Pi = \text{lcm}(\{p^{-1} \mid p \in P^{spec}\})$$

Intuitively, the hyper-period describes the duration until the schedule repeats its entries. With this concept, we define a finite schedule dl, which the monitor repeats every hyper-period. Beside the deadline array we additionally define an offset array off, which stores the time difference between two consecutive deadlines.

Example 3.1.7 (Schedule)**.** Consider the following specification:

```
input a : Int8
output b : Int8 @4Hz := a.hold().defaults(to: 0) + 2
output c : Int8 @2Hz := b + 3
output d : Int8 @5Hz := a.aggregate(over: 2s, using: ∑)
```

In this example the output stream b is evaluated every 250ms, the output stream c every 500ms, and the output stream d every 200ms. This results in the hyper-period $\Pi = 1\text{s}$, in the schedule dl, and in the offset array off:

$$\text{dl}(0) := (200\text{ms}, \{d\}) \qquad \text{off}(0) := 50\text{ms}$$
$$\text{dl}(1) := (250\text{ms}, \{b\}) \qquad \text{off}(1) := 150\text{ms}$$
$$\text{dl}(2) := (400\text{ms}, \{d\}) \qquad \text{off}(2) := 100\text{ms}$$
$$\text{dl}(3) := (500\text{ms}, \{b, c\}) \qquad \text{off}(3) := 100\text{ms}$$
$$\text{dl}(4) := (600\text{ms}, \{d\}) \qquad \text{off}(4) := 150\text{ms}$$
$$\text{dl}(5) := (750\text{ms}, \{c\}) \qquad \text{off}(5) := 50\text{ms}$$
$$\text{dl}(6) := (800\text{ms}, \{d\}) \qquad \text{off}(6) := 200\text{ms}$$
$$\text{dl}(7) := (1000\text{ms}, \{b, c, d\}) \qquad \text{off}(7) := 200\text{ms}$$

△

Finite Memory Handling

Section 3.1.1 presents the types of stream expressions, including different lookups and function calls. The monitor accesses stream values to evaluate the expressions, which needs to be stored. A naive implementation would store every computed or received value to guarantee that the expression evaluation has access to the values. This approach is sane assuming infinite memory. However, in practice this assumption does not hold, and the size of the input data is necessary to guarantee the absence of buffer overflows. Otherwise the monitor loses its guarantees which is critical if monitoring is used as the safety component of a system. Again the assumption of knowing the size of the input data cannot be applied in practice, such that we need another approach, which identifies the memory consumption statically based on the specification. To realize an approach with finite memory, we first assume specifications without sliding window lookups.

→Sec. 3.1, P. 7

The syntax statically encodes which values will be addressed by the stream expression. Thus, we can analyze for each stream the *storage requirement* from the specification, which contains the number of stream values which will be accessed. For illustration, consider the following example which contains all possible lookups:

Def. Storage Requirement

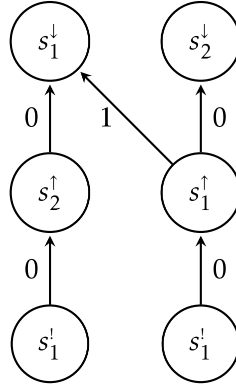


Figure 3.2.: Dependency graph for the specification in Example 3.1.9.

Example 3.1.8 (Storage Requirement)**.** Consider the following specification:

```
input a : Int8
input b : Int8
output c : Bool := a + b.offset(by:-1).defaults(to: 0) < 3
output d : Bool @1Hz := a.hold().defaults(to:1) < 3
trigger c "c is smaller than 6"
trigger d "a is smaller than 3"
```

In this specification, a is accessed with a synchronous lookup and a sample & hold lookup. In both cases, the expression targets the current value of the stream and the monitor does not need to store any previous values. The input stream b is accessed with an offset lookup of −1. Therefore, the current stream value and the previous value need to be stored. The output streams c and d are synchronously accessed by the triggers, such that the storage requirement for both streams is 1. Triggers cannot be addresses by stream expressions resulting in no storage requirement. △

The information about the storage requirement per stream is encoded in the dependency graph, resulting in the following definition:

Definition 15 (Storage Requirement [25])

The storage requirement $\kappa(s)$ of stream s is the maximum offset of stream lookups with target s based on the dependency graph $DG = (V, E)$.

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

Example 3.1.9. Figure 3.2 represents the dependency graph of the specification in Example 3.1.8. It represents that the storage requirement corresponds to the maximal offset of the incoming edges in the dependency graph:

$$\kappa(s_1^{\downarrow}) = 1 \qquad \kappa(s_2^{\downarrow}) = 2 \qquad \kappa(s_1^{\uparrow}) = 1 \qquad \kappa(s_2^{\uparrow}) = 1$$

△

In the next step, we now include sliding window accesses again, starting with the following specification:

```
input a : Int8
output b : Int8 := a.aggregate(over: 1s, using: sum)
```

The sliding window in the output stream b adds up all input values during the last second. Once again, a naive implementation would store all incoming events $a_1, \dots, a_n$ and adds up all values, i.e. $\sum_{1 \leq i \leq n} a_i$. This approach is sane and the sliding window would return the correct value. However, this approach infers two problems. First, with every new event the function adds up all values, e.g. with a new event a_{n+1} the windows computes $\sum_{1 \leq i \leq n+1} a_i$. However, most of the values are already computed with the previous event so a better approach would be re-use the old result $\sum_{1 \leq i \leq n} a_i$, resulting in $(\sum_{1 \leq i \leq n} a_i) + a_{n+1}$. Another problem with storing all incoming values is the memory consumption. The monitor has no restrictions on how many events can be received during the window. Therefore, if we store all events we would need infinite memory. To solve this issue, we restrict the aggregation functions to *list homomorphisms*. With this limitation, we re-use computations with a constant runtime and constant memory.

Definition 16 (Homomorphism [26])

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

Def. List Homomorphism

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

Example 3.1.10 (Add List Homomorphism)**.** The list homomorphism for the addition iteratively adds up all incoming events and is defined as:

- $map_{sum} : A \to A$ with $map_{sum}(x) = x$
- $fin_{sum} : A \to A$ with $fin_{sum}(x) = x$
- $\circledast_{sum} : A^2 \to A$ with $x_1 \circledast_{sum} x_2 = x_1 + x_2$
- $\varepsilon_{sum} = 0$

△

With the following theorem, we can iteratively pre-aggregate the intermediate values, store them, and can re-use the values for following computations. For example, to compute the n+1-th value $v_{n+1} = \text{fin}(\text{map}(a_1) \circledast_\gamma \ldots \circledast_\gamma \text{map}(a_{n+1}))$, we can use the pre-aggregated values $i_{1\ldots n} = \text{map}(a_1) \circledast_\gamma \ldots \circledast_\gamma \text{map}(a_n)$ as input for the finalization function, resulting in $v_{n+1} = \text{fin}(i_{1\ldots n} \circledast_\gamma \text{map}(a_{n+1}))$.

Theorem 3 (Meertens [26])**.** *The aggregation of $v_1, \ldots, v_n$ using a list homomorphism γ can be broken into arbitrary sub-aggregations. Let $(I_i)_{i \leq k} = ((x_{i,j})_{j \leq |I_i|})_{i \leq k}$ for some $k \in \mathbb{N}$ be an ordered partition of the interval* $[1, \ldots, n]$.

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

Because we can re-use the results in the intermediate representation, the sliding window can be pre-computed. For this reason, the evaluation of sliding window expression is performed with constant memory in the size of the intermediate representation. In the previous example, the intermediate representation is a single value. However, this is not true in general shown with the next example:

Example 3.1.11 (Integration List Homomorphism)**.** For the integration function, the input values are tuples $(v, t) \in V \times T$, where v is the current value and t is the corresponding time stamp. To compute the integral, we use the trapezoid construction, a numerical approach represented in Figure 3.3. This approach constructs from samples of the function trapezoids and calculates the area of the trapezoids. With a decreasing difference between the sample, the trapezoids gets more accurate and the approximated value corresponds more to the concrete integral. This results in the following list homomorphism:

- $map_\int : (A, T) \to Optional(A, T, A, T, A)$ with $map_\int((x, t)) = (x, t, x, t, 0)$
- $fin_\int : Optional(A, T, A, T, A) \to Optional(A)$ with
  $fin_\int(\bot) = \bot$ and
  $fin_\int((x^L, t^L, x^R, t^R, v)) = v$

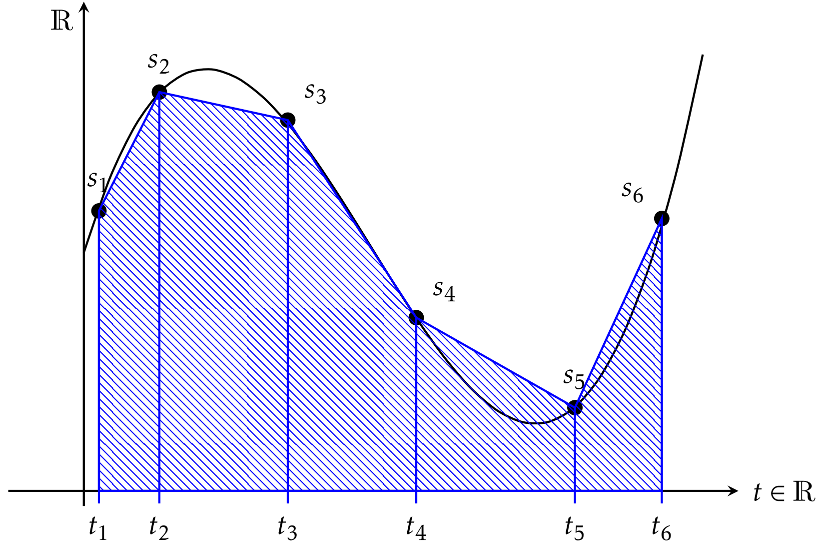


Figure 3.3.: Illustration of the trapezoid construction for integration. The black line describes the underlying function with the sample points $s_1, \ldots, s_6$ for the trapezoid. The numeric approach computes the areas between the samples, represented with the blue lines.

- $\circledast_\int : (Optional(A,T,A,T,A))^2 \to Optional(A,T,A,T,A)$ with
  $\bot \circledast_\int \bot = \bot$,
  $(x_1^L, t_1^L, x_1^R, t_1^R, v_1) \circledast_\int \bot = (x_1^L, t_1^L, x_1^R, t_1^R, v_1)$,
  $\bot \circledast_\int (x_2^L, t_2^L, x_2^R, t_2^R, v_2) = (x_2^L, t_2^L, x_2^R, t_2^R, v_2)$ and
  $(x_1^L, t_1^L, x_1^R, t_1^R, v_1) \circledast_\int (x_2^L, t_2^L, x_2^R, t_2^R, v_2)$
  $= (x_1^L, t_1^L, x_2^R, t_2^R, \frac{1}{2} \cdot (x_1^R + x_2^L) \cdot (t_2^L - t_1^R) + v_1 + v_2)$
- $\varepsilon_\int = \bot$

In the list homomorphism, each intermediate value represents a certain area of the trapezoid. Intuitively, an intermediate value stores its start and end values as well as the current volume. The neutral element is $\bot$, which means that now value is received and no area is not defined. △

Up to this point, we ignored that a sliding window only aggregates stream values inside a certain duration. For illustration, consider the example in Figure 3.4. The points on the black line represent an update of a. When a new event arrives and the output stream b is evaluates, the window expression adds up all values in the last second. In the figure, the windows are represented with the blue lines. We see that the window contains all received values for the first three events. To compute the window, the aggregation function can pre-aggregate them as described previously. However, with the arrival of the fourth event at time 2.2s, the first event at 0.75s is not part of the window

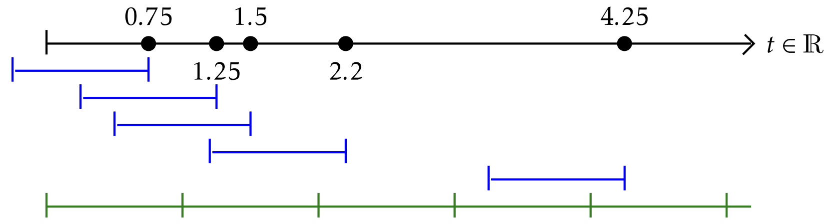


Figure 3.4.: Illustration of sliding windows with event-based and periodic activation conditions. The black line represents the real-time axis, with incoming events. The blue lines describe the windows with an event-based activation condition. The green lines represent the windows with a periodic activation condition.

anymore and therefore not part of the window evaluation. To decide if an event is part of a window, a naive approach annotates each event with its timestamp. As a consequence this approach then stores each event individually and so requires infinite memory. To solve this issue, we consider the inference rule for sliding windows from Section 3.1.3. This rule restricts the activation condition of sliding window expressions to periodic types, such that the evaluation is performed at fixed timestamps. In Figure 3.4, this is represented with the green line for the following specification:

→Sec. 3.1, P. 17

```
input a : Int8
output b : Int8 @1Hz := a.aggregate(over: 1s, using: sum)
```

For the evaluation of the sliding window, the concrete timestamp of a stream value is irrelevant, e.g. the value, which arrives at 1.25s behaves equivalent to the event at 1.5s. Both events are assigned to the same window, without knowing the concrete timestamp. Note that the list homomorphism is not commutative and the order of the events has still an impact on the window evaluation, and the events behave equivalent with respect to their arrival time. With this approach, the expression pre-aggregates its value for each window, e.g. every second separately, and forgets after the evaluation of b its value, because they are not used for further evaluations. For this, we can realize the window expression with finite memory.

In the previous example, the duration δ and period π^{-1} of the frequency is assigned to 1s. For this reason, an incoming event was only part of one window. Figure 3.5 shows the difference with the red lines, if we change the specification to:

```
input a : Int8
output b : Int8 @1Hz := a.aggregate(over: 3s, using: sum)
```

We see that the events at 1.25s and 1.5s are part of three windows: w_1, w_2, and w_3. With the previous approach, we assign both event to three windows and each value pre-aggregates them individually. However, a better approach would be a pre-aggregation of the values between every second and assign those values to the corresponding windows. In the figure this is represented with the dotted lines. This approach is called the bucketing approach proposed by Li et al. [27]. We separate the window in equal size

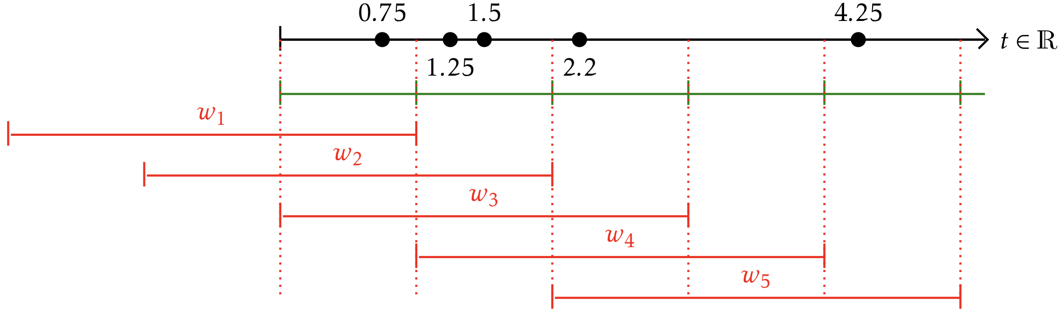


Figure 3.5.: Illustration of sliding windows of different sizes. The black line represents the real-time axis, with incoming events. The green lines describes the windows with duration of one second. The red lines represent the windows with a duration of three seconds, separated in three buckets. The different buckets are illustrated with the dotted lines.

buckets, where each bucket pre-aggregates with the intermediate representation the incoming events. With the arrival of a deadline, we then combine the corresponding buckets of the window and finalize the result. The number of buckets can be determined statically, resulting in finite memory:

Definition 17 (Number of Buckets)

Consider the window $w = Window(s_j^-, \delta, \gamma)$ for a homomorphism $\gamma\colon A^* \rightarrow B$ with mapping $map_\gamma\colon A \rightarrow T$ and $\pi, lift(B) \models w$ for some frequency π. The number of buckets $buckc_w$ and the time of each bucket $buckd_w$ is then:

$$buckc_w := \frac{\operatorname{lcm}(\delta, \pi^{-1})}{\pi^{-1}} \qquad buckd_w := \gcd(\delta, \pi^{-1})$$

To get an intuition for the bucketing approach consider the following example.

Example 3.1.12. Consider the previous specification with a sliding window duration of 3s. For this specification, the number of buckets is $buckc = 3$ and the time per bucket is $buckd = 1$s. Table 3.6 represents the events illustrated in Figure 3.5 with their values, the bucket entries for each relevant timestamp, and the value for the output stream b. The bucket b_3 represents the "newest" bucket whereas b_1 represents the "oldest" one. At the beginning, all buckets are assigned to the neutral element ε. In the list homomorphism for addition the neutral element is 0. With the first event at timestamp 0.75, the newest bucket gets updated with the binary reduction function after lifting the incoming event: $b_3 \circledast_\gamma map_\gamma(5) = 0 + 5 = 5$. When the deadline at 1s is due, the window combines the buckets and finalizes the result, which is the value of b: $fin_\gamma(b_1 \circledast_\gamma b_2 \circledast_\gamma b_3) = 0 + 0 + 5 = 5$. Because the last bucket is now outdated, we shift all bucket entries to the left and initialize a new bucket with the neutral element. At the timestamp 1.25s, the last bucket gets updated: $b_3 \circledast_\gamma map_\gamma(2) = 2$. The same

Event	Time	a	b_1	b_2	b_3	b
	0.0s		ε	ε	ε	
1	0.75s	5	ε	ε	5	
	1.0s		ε	ε	5	5
2	1.25s	2	ε	5	2	
3	1.5s	4	ε	5	6	
	2.0s		ε	5	6	11
4	2.2s	10	5	6	10	
	3.0s		5	6	10	21
	4.0s		6	10	ε	16
	4.25s	1	10	ε	1	

Table 3.6.: Detailed computation of a sliding window.

happens with the next event: $b_3 \circledast_\gamma map_\gamma(4) = 2 + 4 = 6$. With the following deadlines, the aggregation function is finalized. Note that because of the shift, the oldest bucket gets always evicted and therefore also the outdated events. Additionally, the newest bucket has always the same position. △

Remark 3.1.10 (Formal Semantics for Finite Memory). *With the definition of storage requirement and the bucket approach for sliding windows, Schwenger [25] defined a finite memory model and proved that the memory access function has an equivalent behavior for the infinite and the finite memory accesses. In this thesis, we use this result to realize a hardware-based monitor with static memorization bounds.*

With the described approach for the offset handling with finite memory and sliding windows using the bucket approach we define the following theorem:

Theorem 4 (Finite Memory Monitoring [25]). *A valid* RTLola *specification can be accurately evaluated with finite memory.*

Note, from this point we use RTLola with the syntactic suggar proposed by Schwenger [25].

3.2. Hardware Compilation

Baumeister et al. [14] introduced a hardware-based realization for RTLola specifications. The approach compiles a specification into an architecture monitoring incoming

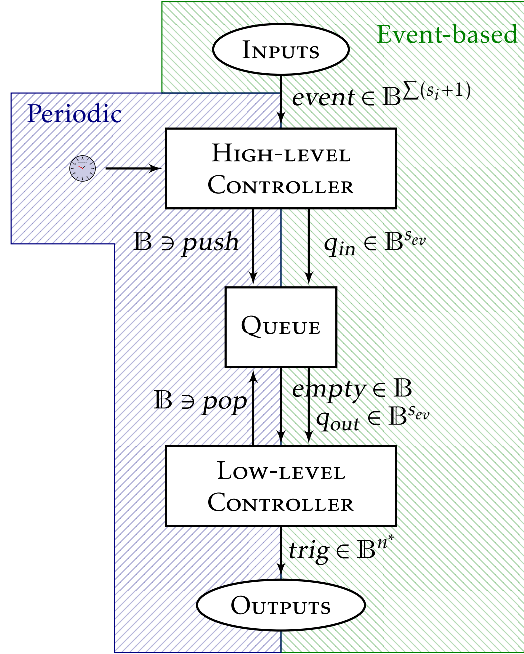


Figure 3.7.: General structure of the hardware-based monitor [14].

sequences with the semantics of RTLola. In Chapter 4, we implement this approach, →Chap. 4, P. 53 compiling the specification into the hardware description language VHDL. The focus of this description is the realization of the same timing behavior of a specification with real-time constraints. For this reason, stream evaluations as well as updating or requesting a sliding window are represented with the functions *eval*, *map*, $\circledast$, and *fin*.

This section has the following structure. At first, we discuss the overall concept of the realization, which is separated into the High-level Controller and Low-level Controller. Then we introduce some further notation before we describe the realization of each component.

3.2.1. General Structure

Figure 3.7 represents the general structure of the hardware-based realization. It consists of two modules that are connected by a First-In-First-Out queue. The separation results from the difference between the evaluation of a stream and its activation. High-level controller (HLC) satisfies the realization of this condition, which differentiates between event-based and periodic streams. To decide if a stream needs to be activated, the HLC computes the system time for periodic streams and receives the input of the monitor for event-based streams. The information, which streams needs to be evaluated and the data from the incoming event, is pushed to the Queue. Therefore, the output of the HLC s_{ev} is defined as $(\sum_{i=1}^{n^{\downarrow}}(s_i+1)) + s_{ts} + n^{\uparrow}$, where:

- The first $(\sum_{i=1}^{n^{\downarrow}}(s_i + 1)) + s_{ts} + n^{\uparrow}$ bits encode the incoming event to the monitor. An incoming event contains for each input stream $s_i^{\downarrow}$, a bit indicating if the current event updates $s_i^{\downarrow}$, and s_i bits containing the data bits. The number of bits s_i and its representation is encoded in the value type.
- The next s_{ts} denotes the timestamp, where either an event was received or if the activation condition of a periodic stream is fulfilled. These timestamps are needed by the sliding window.
- The last $n^{\uparrow}$ bits encode for each output bit if the activation condition is fulfilled in the pushed evaluation cycle or not.

Then, the second component — called Low-level Controller (LLC) — uses the output to preform the evaluation process. Because the output encodes the information which streams are evaluated, the LLC does not differentiate between periodic and event-based streams anymore. This process includes the update of input streams, sliding windows, as well as the evaluation of the stream expressions of output streams. As explained in Section 3.1.2, the order of the stream execution is essential. The LLC contains besides the stream evaluation, a state machine, coordinating the evaluation. The Queue then coordinates the communication between activation and evaluation.

→ Sec. 3.1, P. 14

Remark 3.2.1 (Clock Frequencies)**.** *Because the complexity of deciding if a stream activation condition is fulfilled is lower than performing the update, the* HLC *and* LLC *use different clocks. To handle incoming bursts for a short amount of time, the* HLC*, which receives the incoming events, ticks faster than the* LLC*. During these bursts, the* Queue *acts like a buffer, such that the* LLC *can complete the evaluation with a slower frequency without losing events. Of course, this approach only works for a sudden burst and not a continuous one.*

3.2.2. Notation

In this thesis, we use the following notation:

Definition 18 (Noatation [14])

Let x be a bit string of size n, then:

- $\circ$ is defined as a binary operator, which concatenates two bit strings
- $x[i]$ accesses the ith bit of x, assuming $i < n$
- b^n defines a bit string x, with $\forall 0 < i < n. x[i] = b$
- a substring $x[l \dots u]$ is defined as $x[l] \circ x[l+1] \circ \dots \circ x[u-1]$, excluding $x[u]$ and assuming $l < n \land u \leq n$
- a substring $x[l \dots]$ is defined as $x[l \dots n]$, assuming $l < n$
- a substring $x[\dots u]$ is defined as $x[0 \dots u]$, assuming $u \leq n$

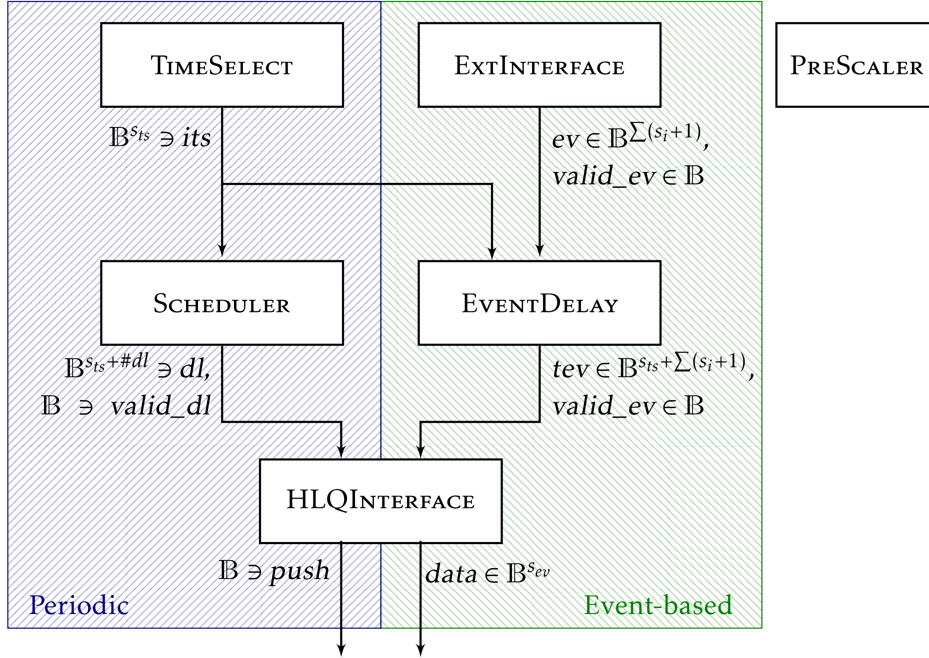


Figure 3.8.: Representation of the High-level Controller [14].

- ξ is defined as the internal clock rate
- $\sum(s_i+1)$ is used for $\sum_{i=1}^{n^{\downarrow}}(s_i+1)$

In the following sections, we distinguish between registers and signals. Signals are written in the italic font and describe the virtual wires between the components. Therefore, signals can be either the input or output of a component. Registers, corresponding to flip-flops, and update their values in each clock cycles. A component uses a register to store a value for the next clock cycle. To access the first value, we assign each register to a default value. To present the complete semantics of the hardware-realization, we describe the assignment of each output signal and register in each clock cycle t.

3.2.3. High-level Controller

The High-level Controller (HLC) receives the incoming events of the monitor, computes the system time, and realizes the activation condition of each stream. Independent of the specification, the HLC always consists of the same components, visualized in Figure 3.8. Before we go into detail through the single components, we first discuss the general concept.

The HLC is designed as a pipeline architecture, which is synchronized by a shared clock. The activation condition of a stream differentiates between periodic and event-based stream, introduced in Section 3.1.3. This behavior is reflected in the architecture →Sec. 3.1, P. 17

with the two sides, event-based and periodic. Periodic streams, declared with a frequency, are evaluated at fixed points in time. For this reason, one part of the HLC is finding these time points, represented with the periodic side. The periodic side consists of the TimeSelect, computing the system time out of the frequency, and the Scheduler, finding the fixed time points.

Event-based output streams are evaluated at non-fixed time steps, represented with the event-based side. The specification bounds their activation condition of an event-based output stream at the incoming event. For this reason, receiving incoming events is covered on this side with the ExtInterface. The incoming events and the system time are forwarded to the EventDelay, which annotates the event with this timestamp. The HLQInterface receives the output from the EventDelay and the Scheduler, decide which streams need to be activated, and forwards this information to the Queue.

Remark 3.2.2. *We describe the semantics of each component individually to describe the behavior of the monitor in general. We define the assignment of each signal and register in each clock cycle and use the input signals described in the corresponding figures.*

Time Select

The TimeSelect computes the system time from the input clock frequency. For this computation, the component uses an internal register **reg_its**, storing the current system time, starting with zero. In each clock cycle, the component adds the internal clock rate ξ to the previous value, resulting in the system time. The output wire *its* contains the same value as the register, which produces no delay.

Definition 19 (TimeSelect [14])

Def. TimeSelect

The TimeSelect component is defined as:

$$\mathbf{reg_its}^0 = 0^{s_{ts}}$$

$$\mathbf{reg_its}^{t+1} = \mathbf{reg_its}^t + \xi = (t+1) * \xi$$

$$its^t = \mathbf{reg_its}^t$$

Scheduler

The task of the Scheduler is to find the fix points in time when the periodic streams are evaluated. First, we declare a bit vector with the register **did**. This bit vector is a unary encoding, for the current position in the hyper-period. This position is corresponds to a deadline (Section 3.1.4). In the first clock cycle, we assign the last position to 1, representing the first deadline. If a deadline is due, we perform a cyclic shift by one position. This shift represents the next deadline in the hyper-period. Because the hyper-period is probably smaller than the execution time and we need to find all critical points in time for period streams, we declare the register **period**. This register

→ Sec. 3.1, P. 24

is initialized with 0. In case that the system time is greater than the hyper-period, we add the hyper-period to the previous value of **period**. This creates the invariant that the difference between this register and the system time *its* describes the current position in the hyper-period. To represent the arrival of a deadline, we use the progress signal *prog*. This register compares the current position of the hyper-period against the entry in the static defined deadline array dl. This array contains for each deadline **did** an entry $o_{\mathbf{did}}$, such that $o_{\mathbf{did}}$ is the duration until deadline **did** is due in the hyper-period. With this approach, we can find all relevant periodic timestamps with finite memory.

Definition 20 (Scheduler [14])

The Scheduler is formally defined as:

Def. Scheduler

$$init^t = \begin{cases} 1 & \text{if } t = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathbf{did}^0 = 0^{\#dl}$$

$$\mathbf{did}^{t+1} = \begin{cases} 10^{\#dl-1} & \text{if } init^{t+1} \\ \text{csr}(\mathbf{did}^t) & \text{if } \neg init^{t+1} \land prog^{t+1} \\ \mathbf{did}^t & \text{otherwise} \end{cases}$$

$$\mathbf{period}^0 = 0^{\#dl}$$

$$\mathbf{period}^{t+1} = \begin{cases} 0 & \text{if } init^{t+1} \\ \mathbf{period}^t + \Pi & \text{if } \mathbf{did}^t = 0^{\#dl+1}1 \land prog^{t+1} \\ \mathbf{period}^t & \text{otherwise} \end{cases}$$

$$prog^{t+1} = \mathbf{did}^t \neq 0^{\#dl} \land (its^{t+1} - period^t) > \text{dl}(\mathbf{did}^t)$$

$$dl^t = its^t \circ \mathbf{did}^t$$

$$valid_dl^t = \neg prog^t$$

In the definition, csr is a function which performs a cyclic shift, and dl is the statically defined offset array.

ExtInterface

The ExtInterface receives the incoming event to the monitor. To decide if the current data on **data_in** is a new event, the monitor and the monitored system use a shared register **avail**. If the system assigns the register to 1, the ExtInterface forwards the data stored in **data_in** to the EventDelay. Additionally, the component clears the **avail** bit,

indicating that a new input event can be received. To forward an event, the component assigns the output signal *ev* to **data_in** and to zero otherwise. To signalize a valid event, i.e. a component interprets the values on the wires as an event, we use the signal *valid_ev*. This signal is assigned to the **avail** storing this information.

Definition 21 (ExtInterface [14])

Def. ExtInterface

The ExtInterface is defined as:

$$ev^0 = 0^{\sum s_i}$$

$$ev^{t+1} = \begin{cases} \mathbf{din}^t & \text{if } \mathbf{avail}^t \\ 0^{n^{\downarrow}+n^{\uparrow}} & \text{otherwise} \end{cases}$$

$$\mathbf{avail}^0 = 0$$

$$\mathbf{avail}^{t+1} = \begin{cases} 1 & \text{if } external^t \wedge \neg \mathbf{avail}^t \\ 0 & \text{otherwise} \end{cases}$$

$$valid_ev^0 = 0$$

$$valid_ev^{t+1} = \mathbf{avail}^t$$

In the definition, the signal $external^t$ is an oracle. It represents the input of the monitored system if the current data expresses a new event.

EventDelay

The EventDelay annotates the forwarded event with the current system time computed by the TimeSelect. The LLC uses the time annotation to decide which stream values are part of which window. Therefore, we assign the output signal to the input signals.

Definition 22 (EventDelay [14])

Def. EventDelay

The EventDelay is defines as:

$$tev^t = valid_ev^t \circ its^t \circ ev^t$$

HLQInterface

The HLQInterface unions the periodic and the event-based side of the HLC. For this reason, it receives the ID of the current deadline as well as the annotated event. In both cases, the input contains the corresponding timestamp. The component then decides which streams are affected by the current data. Afterward, it pushes this information

as well as the input data to the queue. The union between the periodic side and the event-base side results in a problem, because of the possibility that a deadline, as well as an event, arrives in the same clock cycle of the HLC. The HLQInterface is only able to push one input value to the Queue in one clock cycle. However, if we unite the information from the Scheduler and the EventDelay, we change the semantics of the monitor realization. If the component only pushed the information caused by the deadline, the information about the event is lost in the next clock cycle, which also changes the semantics. To solve this issue, we change the clock frequency of the HLQInterface, which ticks twice as fast as the other components in the HLC. With this adaption, the HLQInterface pushes the event and the deadline separately, without losing any information.

The output of the HLQInterface is divided into the data bits and the push bits. To present the output signal assignment formally, we separate the tasks in event and odd clock cycles.

In event clock cycles, the HLQInterface interprets the output of the EventDelay. For this, it pushes the value on the data signal to the queue iff the input is *valid_ev* is 1. The first *tev* bits represent the event from the EventDelay. The last bits assign the output streams to their activation condition results. We use for event-based clock cycles a static array dep of size $n^{\downarrow}$, where each element $\text{dep}(i)$ is of size $n^{\uparrow}$. These elements encode the dependencies between output streams and input streams, i.e. the bit $\text{dep}(i)(j)$, is assigned to one, iff the output stream $s_j^{\uparrow}$ has a transitive dependency to $s_i^{\downarrow}$. The HLQInterface combines the encoding with the current input, resulting in the bit representation in which output streams need to be evaluated.

Definition 23 (HLQInterface (Even Clock Cycles) [14])

Even clock cycles of the HLQInterface are defines as:

Def. HLQInterface Even Clock Cycle

$$push^t = valid_ev^t$$

$$data^t = ev^t \circ \bigwedge_{i=1}^{n^{\downarrow}} \left(\neg\,\text{dep}(i) \vee \left(ev^t \Big[\sum_{j=1}^{i} (s_j + 1) - 1 \Big] \right)^{n^{\uparrow}} \right)$$

The static array dep is defined as:

$$\text{dep}(i)(j) = \begin{cases} 1 & \text{if } s_i^{\downarrow} \in dep_{s_j^{\uparrow}} \\ 0 & \text{otherwise} \end{cases}$$

Intuitively, the combination encodes the following constraint: If a stream $s_j^{\uparrow}$, depends on input stream $s_i^{\downarrow}$, then $s_j^{\uparrow}$ is evaluated if $s_i^{\downarrow}$ is contained in the current event. Because $s_j^{\uparrow}$ might depend on several inputs, we need to check if all streams are updated. For clarification, we show an example:

Example 3.2.1 (Encoding of the Deadline Array)**.** Consider the following example:

```
input a : Int8
input b : Int8
output c : Int8 := a + 3
output d : Int8 := b + 4
output e : Int8 := e + d
```

The specification results in the dependency array dep:

- $\text{dep}(s_0^{\downarrow}) = 101$
- $\text{dep}(s_1^{\downarrow}) = 011$

With the input updating only a, the realization combines:

$$\begin{aligned} &(\neg \text{dep}(s_0^{\downarrow}) \vee 111) \wedge (\neg \text{dep}(s_1^{\downarrow}) \vee 000) \\ =&(010 \vee 111) \wedge (100 \vee 000) \\ =&100 \end{aligned}$$

△

Remark 3.2.3. *In the original encoding, there is a bug in the combination between* dep *and ev. In this thesis, this bug is fixed by the previous assignment.*

In odd clock cycles the HLQInterface interprets the output of the Scheduler. Because input streams are always event-based, the data bits for the input events are assigned to zero. The component inherits the timestamps from the Scheduler. To decide which periodic streams the evaluation cycle updates, the realization uses the lookup table dl_target, which takes as input the current deadline id, and returns an encoding of the affected streams.

Definition 24 (HLQInterface (Odd Clock Cycles) [14])

Def. HLQInterface Odd Clock Cycle

Even clock cycles of the HLQInterface are defined as:

$$push^t = valid_dl^t$$

$$data^t = 0^{\sum(s_i+1)} \circ dl^t[\ldots s_{ts}] \circ \text{dl_target}(dl^t[s_{ts} \ldots])$$

PreScale

The previous paragraph explains the need for two clocks in the HLC. The PreScale realizes this logic and computes a slower *sclk* clock out of the *hclk*. The architecture assigns the *sclk* to the Scheduler, the ExtInterface, and the EventDelay. The HLQInterface uses the faster clock *hclk*. The TimeSelect component uses the system clock to compute the system time. Figure 3.8 does not present these wires for better readability.

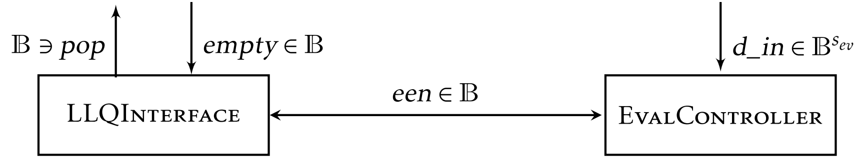


Figure 3.9.: Representation of the general structure of the Low-level Controller [14].

Remark 3.2.4 (Online / Offline Monitoring)**.** *Baumeister et al. [14] describe a hardware realization, which is capable of offline and online monitoring. Intuitively, the difference between online and offline monitoring is the used timestamp. Online monitoring is an analysis during the execution of the monitored system. As timestamps, the monitor uses the system time and assumes that between two clock cycles of the* HLC *at most one deadline is reached. Offline monitoring is a post-run analysis, where the monitored system produces a log file during its execution. To validate the timing constraint, the monitor used the time stamps annotated in the log file. Because the monitor does not receive a continuous-time signal, the previous assumption is not appropriate. As a consequence, the* HLC *also needs to handle two consecutive deadlines between two incoming events. This results in a buffering in the* EventDelay *component. Because the overall goal is the integration of a hardware board monitoring a UAV during its execution, we do not introduce the realization of offline monitoring.*

3.2.4. Low-level Controller

The Low-level Controller (LLC) pops the elements from the queue and evaluates the streams with the current input. Figure 3.9 represents the general architecture of the LLC. In general, the LLC consists of the LLQInterface and the EvalController. As the HLQInterface, the LLQInterface coordinates the communication with the Queue. It checks if the Queue has available data and raises the pop signal if the EvalController is ready to receive a new incoming event. The EvalController then realizes the underlying stream evaluation and contains a state machine and for each stream and sliding window a single entity. The state machine enables the stream and sliding window entities based on their evaluation order. These entities wait on this enable signal and perform the updates. Output stream entities in comparison to input stream entities contain two enable signals, one for the pseudo evaluation and one for the expression evaluation. Similar sliding window entities have three input enable signals: The first one, the signal *evict*, enables the check for the affection of the current timestamp on the bucket approach. The second enable signal *upd* update the buckets, with the current input, and the last enable signal *request* requests the result of the aggregation function. A more detailed description of the single entities is presented in the next paragraphs.

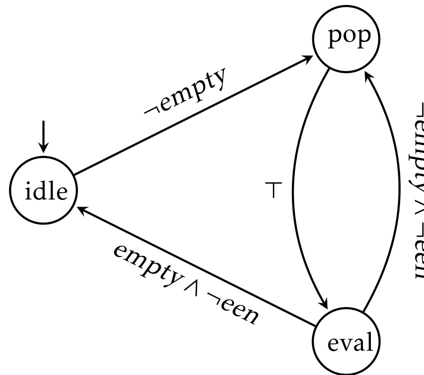


Figure 3.10.: Representation of the state machine in the LLQInterface [14].

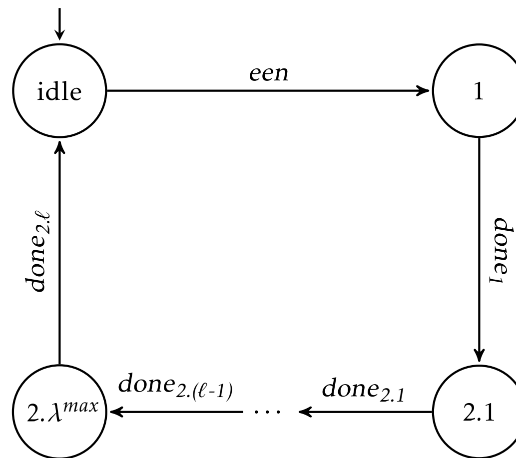


Figure 3.11.: Representation of the state machine in the EvalController [14].

LLQInterface

Figure 3.10 represents the realization of the communication between the LLC and the Queue with a moore machine. Starting with the `idle` state, the LLQInterface pops the elements from the Queue as soon as they are available. To access the value, it checks the *empty* signal and performs the transition to the pop state. In this state the LLQInterface sets the *pop* signal to 1 for one clock cycle and goes to the `eval` state. This state disables the *pop* signal and enables the *een* starting the evaluation cycle of the Evaluator. As soon as the evaluation cycle finishes, the state machine goes to the `idle` state if the Queue is empty and the pop state otherwise. The end of the evaluation cycle is identified by the een signal. The hardware realization of this state machine is a lookup table realizing a function that performs the transitions and output values.

EvalController

The EvalController implements the evaluation order from Section 3.1.2. As the LLQInterface, this component is realized as a Moore state machine, represented in Figure 3.11. In comparison to Figure 3.10, the number of states depends on the specification, more concrete on λ^{max}. The state machine waits on a rising edge of the *een*, set by the LLQInterface. With this transition, the state machine starts the evaluation cycle of the current input. Starting with the state 1, each state realizes one layer in the evaluation. Therefore, the output signals of each state represent the enable signal for the stream and sliding window entities. The first state 1 corresponds to the input update and pseudo-evaluation phase. For this, the output enables all input streams, which are part of the current event, and all output streams satisfying their activation conditions. This information is encoded in the queue output, as described in Section 3.2.3. Additionally, by receiving the current timestamp, the state machine enables the $evict_\eta$ signals of all sliding window entities. Then, the EvalController waits on a rising edge of $done_1$ indicating that each affected entity performed its update. Formally,

→Sec. 3.1, P. 14

Definition 25 (EvalController Layer 0)

The first state 1 assigns the output signals upd_i, pe_j, and $evict_\eta$ to:

Def. EvalController Layer 0

$$\forall i \leq n^{\downarrow} : upd_i = d_{in}[\sum_{n \leq i}(s_n + 1)]$$

$$\forall j \leq n^{\uparrow} : pe_j = d_{in}[\sum(s_i + 1) + s_{ts} + j]$$

$$\forall \eta \leq n^{w} : evict_\eta = 1$$

The signal $done_1$ is assigned to:

$$done_1 = \Big(\bigwedge_{i \leq n^{\downarrow}} upd_i \implies done_i\Big) \wedge \Big(\bigwedge_{j \leq n^{\uparrow}} pe_j \implies done_j\Big) \wedge \bigwedge_{\eta \leq n^{*}} done_\eta$$

Remark 3.2.5. *Note that the used implication ensures that only the done signals of the enable entities need to be set. For the disabled streams, this information is irrelevant.*

The following states 2.1 to 2.λ^{max} implement the rest of the evaluation order. Depending on the current layer x, the state machine starts the evaluation of output streams, the update of a sliding window, and the request of a sliding window. The update of a sliding window has to be performed after the update or evaluation of the target stream. The request signal has to be enabled if the source stream is affected by the current event. Afterward, it waits on the rising edge of $done_x$ before proceeding to the next layer.

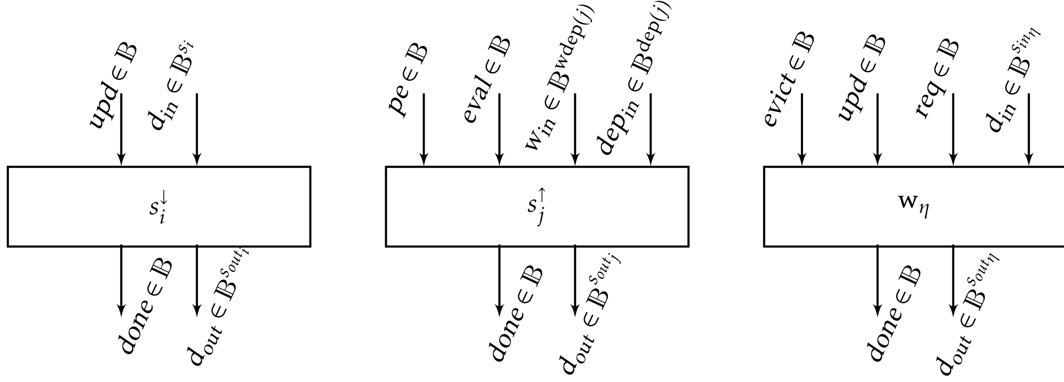


Figure 3.12.: Representation of input stream, output stream, and sliding window components [14].

Definition 26 (EvalController Layer 1 to λ^{max})

Def. EvalController Layer 1 to λ^{max}

The states 2.x assign the output signals $eval_j$, and udp_η to:

$$eval_j = j \in \text{Layer}(x) \wedge d_{in}[\sum(s_i + 1) + s_{ts} + j]$$

$$upd_\eta = d_{out_{\text{tar}(\eta)}}[s_{\text{tar}(\eta)}]$$

$$req_\eta = d_{in}[\sum(s_i + 1) + s_{ts} + s_{\text{src}(\eta)}]$$

The signal $done_x$ is assigned to:

$$done_{2.x} = \Big(\bigwedge_{j \le n^\uparrow} eval_j \implies done_j\Big) \wedge \Big(\bigwedge_{\eta \le n^w} udp_\eta \implies done_\eta\Big) \wedge \Big(\bigwedge_{\eta \le n^w} req_\eta \implies done_\eta\Big)$$

The last transition from $2.\lambda^{max}$ to `idle` ends the evaluation process by disabling *een*.

Input Stream Entities

The general structure of input stream entities is represented in Figure 3.12. In these entities, we store the input values, which are accessed by stream expressions. To store all values, we use $\kappa(s_i^{\downarrow})$ registers, where κ is the storage requirement from Section 3.1.4. With the rising edge of the *upd* signal, the current stream values are shifted, and the input value is set to the current value. To indicate that an offset value is not yet available, we use an additional bit in each register. In the beginning, this bit is assigned to zero, indicating a lookup fail. With each update, this bit is assigned to one. Therefore, with the shift operations, the unavailable values are deleted.

→ Sec. 3.1, P. 24

Definition 27 (InputStreams [14])

With the rising edge of udp_i the input streams perform the following assignments:

Def. InputStreams

$$done^t = upd^t$$

$$\mathbf{R}_n^0 = 0^{s_i+1}$$

$$\mathbf{R}_n^{t+1} = \begin{cases} \mathbf{R}_{n+1}^t & \text{if } upd^{t+1} \wedge n \neq \kappa(s_i) \\ \mathbf{R}_n^t & \text{if } \neg upd^{t+1} \\ {d_{in}}^{t+1} \circ 1 & \text{if } upd^{t+1} \wedge n = \kappa(s_i) \end{cases}$$

$${d_{out}}^0 = 0^{\kappa(i) \cdot (s_i+1)}$$

$${d_{out}}^{t+1} = \mathbf{R}_1^t \circ \cdots \circ \mathbf{R}_{\kappa(s_i)}^t$$

Output Streams Entities

The input and output signals of output stream components are represented in Figure 3.12. An output stream component implements an output stream in the specification. As input stream components, these entities store the values accessed by other streams. However, in comparison to input streams, these values need to be computed with the stream expressions. For this computation, the entity receives for each stream lookup the current value as well as the addition bit, indicating a lookup fail. Window expressions are outsourced in separate components. As a consequence, their values are also received as inputs.

The implementation of output streams is separated into two phases, the pseudo-evaluation (Section 3.1.4), and the evaluation. Both phases are introduced with their corresponding enable signals. As input streams, the pseudo evaluation shifts the offset values. The current value is assigned to a default value $\perp$, which will never be accessed by the output streams, because of the evaluation order (Section 3.1.2).

→Sec. 3.1, P. 24

→Sec. 3.1, P. 10

Definition 28 (OutputStreams Pseudo Evaluation)

With the rising edge on the pe, output stream entities perform the following assignments:

Def. OutputStreams Pseudo Evaluation

$$done^t = pe^t$$

$$\mathbf{R}_n^0 = 0^{\kappa(j) \cdot (s_j+1)}$$

$$\mathbf{R}_n^{t+1} = \begin{cases} \perp & \text{if } n = \kappa(j) \\ \mathbf{R}_{n+1}^t & \text{otherwise} \end{cases}$$

$${d_{out}}^0 = 0^{\kappa(j) \cdot (s_j+1)}$$

$$d_{out}{}^{t+1} = \mathbf{R}_1^t \circ \cdots \circ \mathbf{R}_{\kappa(s_j)}^t$$

The second phase is introduced with a rising edge from the $eval_j$ signal. In this phase, the stream is evaluated based on the evaluation order such that all dependent streams and sliding window lookups are updated beforehand. The result from the stream evaluation replaces the default value $\perp$ from the pseudo evaluation.

Definition 29 (OutputStreams Evaluation)

Def. OutputStream Evaluation

With the rising edge on *eval*, output stream entities perform the following assignments, where the result of the stream expression is represented with evalexpr(j):

$$done^t = eval^t$$

$$\mathbf{R}_n^{t+1} = \begin{cases} \text{evalexpr}(j) \circ 1 & \text{if } eval^{t+1} \wedge n = \kappa(j) \\ \mathbf{R}_n^t & \text{otherwise} \end{cases}$$

$$d_{out}{}^{t+1} = \mathbf{R}_1^t \circ \cdots \circ \mathbf{R}_{\kappa(s_j)}^t$$

Remark 3.2.6. *The evaluation of the stream expression can be separated into several clock cycles to increase the clock frequency. This adaption has an impact on the evaluation order such that the* done *needs to be set after completing the evaluation.*

Sliding Windows Entities

Sliding window lookups are in this hardware-based approach outsourced to separate entities. Recap, sliding windows can be realized with finite memory if we restrict the aggregation function to list homomorphism and use the bucket approach (Section 3.1.4). The input and output signals are represented in Figure 3.12. The realization of sliding windows differentiates three phases, introduced by their enable signals.

→ Sec. 3.1, P. 24

The first phase starts with the rising edge on the *evict* signal. In this phase, the component decides if a bucket is outdated. Therefore, it uses the internal register **T**, holding the timestamp when a new bucket has to be created. The single buckets are represented with the β internal registers **R**, where β is the number of needed buckets. If the current timestamp is greater than the value in **T**, the component creates a new bucket, which is initialized with the neutral element ε and shifts the other buckets. Thereby, the outdated bucket gets deleted. Additionally, the timestamp is updated by adding the period p_η, the inverse of the extend frequency of the source stream, to the old value.

Definition 30 (SlidingWindows Evict)

Def. SlidingWindows Evict

A sliding window component with β buckets and extension period π_η^{-1} performs with a rising edge of the *evict* signal the following assignments:

$$\mathbf{T}^0 = 0^{s_{ts}}$$

$$\mathbf{T}^{t+1} = \begin{cases} \mathbf{T}^t & \text{if } d_{in}[\ldots s_{ts}] \leq \mathbf{T}^t \\ \mathbf{T}^t + p_\eta & \text{otherwise} \end{cases}$$

$$done^0 = 0$$

$$done^{t+1} = d_{in}[\ldots s_{ts}] \leq \mathbf{T}^t$$

$$\mathbf{R}_n^0 = \varepsilon$$

$$\mathbf{R}_n^{t+1} = \begin{cases} \varepsilon & \text{if } n = \beta \wedge d_{in}[\ldots s_{ts}] > \mathbf{T}^t \\ \mathbf{R}_{n+1}^t & \text{if } n \neq \beta \wedge d_{in}[\ldots s_{ts}] > \mathbf{T}^t \\ \mathbf{R}_n^t & \text{if } d_{in}[\ldots s_{ts}] \leq \mathbf{T}^t \end{cases}$$

The second part is introduced if the target stream of the sliding window is updated. In this case, the bucket pre-aggregations also needs to update its value. The decision, when updates are performed, is outsourced to the state machine in the EvalController, which sets the udp_η. With a rising edge of this enable signal, the component updates the last bucket entry with the current input. For this, the input value is lifted to the bucket representation with the map function and concatenated with the current entry using the $\circledast$ reduction.

Definition 31 (SlidingWindows Update)

Def. SlidingWindows Update

A sliding window component with β buckets and extension period π_η^{-1} performs with a rising edge of the *upd* signal the following assignments:

$$\mathbf{R}_\beta^{t+1} = \mathbf{R}_\beta^t \circledast \text{map}({d_{in}}^{t+1})$$

$$done^t = upd^t$$

The last phase is the request phase starting with the rising edge of *req* signal. In this phase, the return value of the aggregation function is computed out of the pre-aggregated buckets. Therefore, we combine all bucket entries with the binary operator $\circledast$ and finalize the result afterward to get the return value.

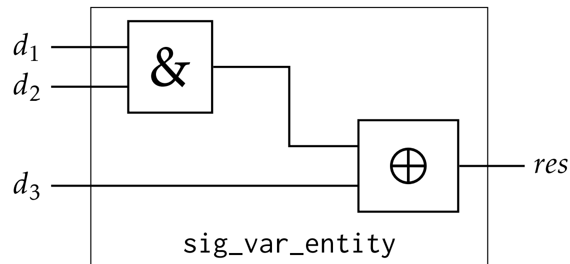


Figure 3.13.: Representation of the circuit described in Section 3.3.1

Definition 32 (SlidingWindows Request)

Def. SlidingWindows Request

A sliding window component with β buckets and extension period π_η^{-1} performs with a rising edge of the *req* signal the following assignments:

$$d_{out}{}^{t+1} = \text{fin}(\mathbf{R}_1^t \circledast \cdots \circledast \mathbf{R}_\beta^t)$$

$$done_{2.x}^t = rq^t$$

3.3. VHDL & FPGA

`VHDL` or `VHSIC-HDL` (Very High Speed Integrated Circuit Hardware Description Language) is a hardware description language to describe logic circuits. For this, the language describes the abstract behavior of the different digital components and their connections. `VHDL` allows the developer to simulate the digital circuit and supports the generation of a bitstream, which can be executed on a Field Programmable Gate Array (FPGA). This last process is often called synthesis. Because of the simulation and the execution on FPGAs, `VHDL` is extremely popular in the development of digital circuits. It helps to find bugs in the design before its production, which reduces the cost of its development. Additionally, the research can also analyze the timing behavior of the circuit.

→ Chap. 4, P. 53
→ Sec. 3.2, P. 32

Chapter 4 describes a prototype implementation which compiles an RTLola specification into `VHDL` code using the architecture from Section 3.2. Afterward, the `VHDL` code is synthesized on an FPGA to monitor a system with a hardware-based realization. This thesis can not describe the full complexity of `VHDL`. Nevertheless, we want to give an intuition about the general structure by describing the implementation of an example.

3.3.1. VHDL Example

Consider the design in Figure 3.13. This component has three input signals `d1`, `d2`, and `d3`, and one output signal `res`. The output signal is computed by combining the first two input signals with a logical *and*. Then, we combine the result with the third input signal using the *xor* gate. The computation is described mathematically as: $res := (d_1 \wedge d_2) \oplus d_3$. The boolean operand table of the component is presented in Table 3.15:

```
1  entity sig_var is
2      port(
3          d1, d2, d3: in std_logic;
4          res: out std_logic);
5      end sig_var;
6
7  architecture behv of sig_var is
8
9  begin
10   process(d1,d2,d3)
11     variable var_s1: std_logic;
12   begin
13     var_s1 := d1 and d2;
14     res <= var_s1 xor d3;
15   end process;
16 end behv;
```

(a) Sequential Execution

```
1  entity sig_var is
2      port(
3          d1, d2, d3: in std_logic;
4          res: out std_logic);
5      end sig_var;
6
7  architecture behv of sig_var is
8
9    signal sig_s1: std_logic <= '0';
10 begin
11   process(d1,d2,d3)
12   begin
13     sig_s1 <= d1 and d2;
14     res <= sig_s1 xor d3;
15   end process;
16 end behv;
```

(b) Parallel Execution

Figure 3.14.: VHDL code fragment for the circuit in Figure 3.13.

The realization of this circuit in VHDL is shown in Figure 3.14a. The implementation starts with the keyword `entity` declaring the interface of the component (lines 1 -5). In the first lines, the implementation assigns the component to a unique name `comp`. With this name, other components in the digital design call the now defined component. This allows the developer to split the digital design and the logic into single components, which can be tested independently of each other. Additionally, the interface defines the input and output signals of the entity and their corresponding types. In our example, all signals are of type bool, but VHDL has a static type system, which allows us to assign bit-vectors to their corresponding representation, e.g. a signed or unsigned interpretation. During the simulation, type errors are reported helping to find bugs in the design. The entity declaration is followed by the `architecture`. An architecture is a description of the functionality of the model and contains the declaration of internal signals and processes. Internal signals are usually registers which store their values until the next assignment. In our example, we do not have internal signals, because the computation has to be performed in one clock cycle. Processes implement the logic of the entity by assigning the output signals to their defined values. In our case (lines 13 and 14), we assign the result from `d1 and d2` to a temporary variable and afterward the output signal to `temp xor d3`. Variables are declared after the process keyword and can only be accessed in the current process. In comparison to signal assignments that are assigned parallelly, variable assignments are executed sequentially. The difference between signals and variables is also manifested in the syntax of VHDL: The syntax for a signal assignment is defined as <=, whereas for a variable assignment the syntax is `:=`.

d_1	d_2	d_3	res
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Table 3.15.: Boolean operand table for $res := (d_1 \wedge d_2) \oplus d_3$.

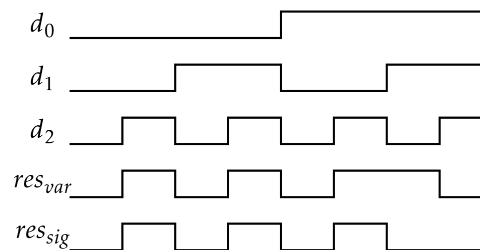


Figure 3.16.: Representation of the execution of the entities in Figure 3.14. A low value of a signal represents the boolean value zero, whereas a high value represents the value one. The signals d_1, d_2, and d_3 are input signals. The signal res_{var} represents the output signal of the entity in Figure 3.14a, and the signal res_{sig} represents the output of the entity in Figure 3.14b.

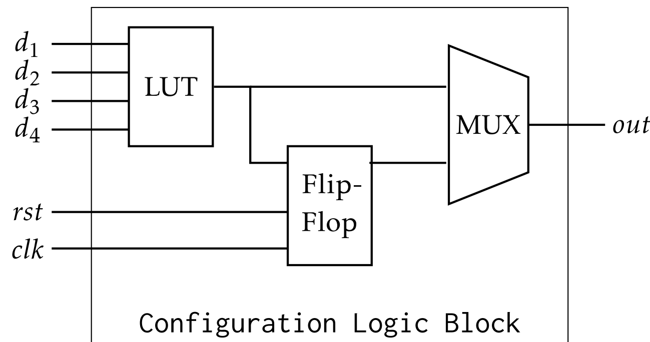


Figure 3.17.: Representation of a CLB in an FPGA

To see the difference, we additionally implement the entity with a signal assignment instead of a variable assignment, shown in Figure 3.14b. Then, we simulate the entities with the input from the boolean operation table. The results are represented in Figure 3.16. We see that the entities differ with the input 110, where the entity with the variable assignments returns the expected value 1 and the other entity 0. Because signal assignments are executed in parallel the output assignment uses the previous result of `temp`, resulting in $temp = 1 \wedge 1 = 1$ and $out = 0 \oplus 1 = 1$.

3.3.2. FPGA

An FPGA is an integrated circuit, which is used to implement the logic of a digital circuit. The logic of the circuit is described by a hardware description language, e.g. VHDL or Verilog. In general, an FPGA consists of I/O pins, an interface to communicate with the environment, configurable logic blocks (CLBs) [28, 29, 30], the interface to program the circuit, and interconnects to connect the CLBs. A CLB contains of a 4- or 6- bit lookup-table, a 1-bit flip-flop and a multiplexer, represented in Figure 3.17.The lookup table contains a boolean operation table, which is the implementation of the design. The flip flops are used to store specific values. Besides the lookup-table, the synthesis of the hardware description language also programs the connections between the CLBs. Based on the model of the FPGA, a CLB has 4 or 6 input bits for the lookup tables. To implement logics with more than 4 or 6 inputs bits, the implementation uses more CLBs. In hardware, all computations have to be performed in one clock cycle, to return the correct result. Therefore, a long depth of a sequential structure slows down the frequency of the computation. To increase the clock frequency, the synthesizer tries to parallelize as many computations as possible. However, the first step of a fast implementation is a parallel design of the hardware circuit.

Chapter 4

Prototype

This chapter describes the general idea of our prototype, which compiles an RTLola specification into synthesizable VHDL code. The implementation is based on the architecture from Section 3.2. To get our resulting hardware monitor, we compile a separate entity out of a template for each element in the hardware description.

→Sec. 3.2, P. 32

The implementation itself is based on the StreamLab framework[1], implemented in the Rust programming language[2]. StreamLab is a monitoring framework that interprets incoming data with respect to an RTLola specification. Figure 4.1 represents the structure of the framework, including the prototype. StreamLab is separated into two parts, the frontend, and the backends. The frontend first generates an intermediate representation (IR) from a given RTLola specification. Part of the IR is the abstract syntax tree (AST). Additionally, it contains the result of different analyzes on the specification, e.g. the memory bound (Section 3.1.4). The interpreter, one of the backends, uses this IR to compute the stream values for the incoming data. Our prototype, the compiler, implements a new backend in the framework. It takes the IR from the frontend as input and compiles for each element in the hardware description a separate VHDL file from a template. These files can be synthesized into an FPGA resulting in a hardware-based monitor. The monitor receives the input data and evaluates the streams.

→Sec. 3.1, P. 24

One advantage of the new backend is the hardware-based approach. If possible, hardware parallelizes computations without producing any overhead in comparison to a software-based solution. Because of the modular structure of RTLola and the design of the approach from Section 3.2, most computations can be parallelized[14], e.g. streams in the same evaluation layer have no execution order allowing for a parallel execution. For this reason, the realization uses the advantage of hardware. Another difference between the two backends is the interpretation and the compilation of the input. The interpreter takes as input the IR and the incoming events and evaluates the stream in the framework. As a result, the specification is only encoded as data. In comparison,

[1] stream-lab.eu
[2] https://www.rust-lang.org

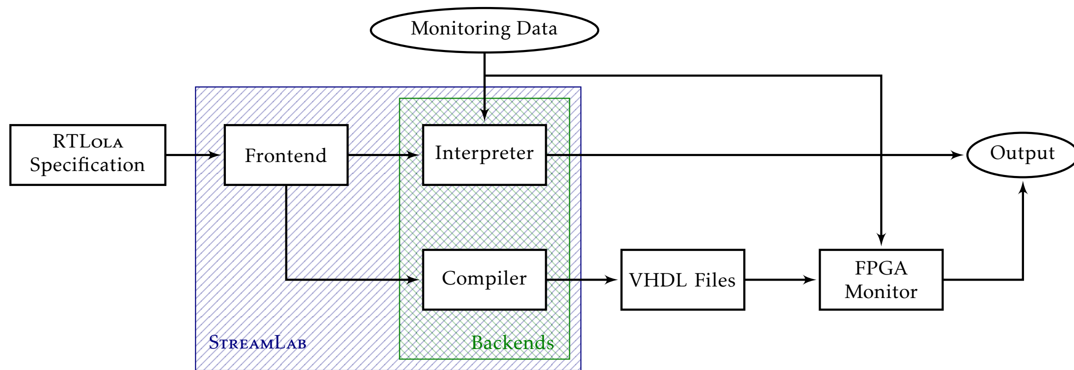


Figure 4.1.: This figure is a representation of the general structure of the StreamLab framework. StreamLab takes an RTLola specification as input. Then, the frontend parses the input specification to an intermediate representation (IR) and forwards this IR to a backend. The interpreter receives the monitored data besides the IR and interprets them with the IR. The interpreter produces output values for each stream in the specification. The compiler receives only the IR from the frontend as input and produces VHDL code. This code can then be synthesized to a monitor, which receives the monitored data and produces the stream values.

the compiler takes as input only the IR and outputs VHDL files realizing the incoming specification. To monitor the incoming events, these files need to be synthesized into an FPGA, which evaluates the streams. With this approach, the specification is encoded in code fragments instead of data fragments that can be described with specification-specific annotations. If the compiler performs these annotations automatically, we call this process a *traceable* compilation.

Traceability

Traceability is part of the requirement-management for system engineering. Intuitively, it describes the relationship between the requirements and the realization during the development phase. Usually, the development starts with defining requirements for the system. These requirements are refined to requirements for the software and hardware, which are afterward implemented and tested. To keep track that the system requirements correspond to the product, we relate each refinement to the previous generalization and vice versa, i.e. the system requirements with the software and hardware requirements, the software and hardware requirements with the resulting code, and the code fragments with the test benchmarks. With this approach, we can ensure that no functionality is missed, but also no dead code or additional functionality is added. We propose an additional step in which we describe the software and hardware requirements as an RTLola specification. A specification is more abstract than code, which simplifies the description of the relation between the components. In a traceable compilations as our prototype, the compiler relates the specification with the code fragments automatically, which produces no additional step. Concerning a hardware-based monitoring approach, industrial companies [31] work on a solution

to solve the certification challenge for FPGA solutions, which includes that up to the hardware design each step needs to be traceable.

In our traceability compilation, we relate the RTLola specification with the compiled `VHDL` code, and vice versa. This process helps to indicate that both encodings have the same semantics. Intuitively, a specification defines requirements, e.g. the evaluation of the stream expression or the activation condition. On the one hand, these requirements need to be realized in the implementation. On the other hand, each code fragment needs to be reflected in the specification to guarantee the same behavior. Depending on the component, the relation between the specification and the realization is direct or indirect. For example, the realization of a stream expression `a + b` has a direct connection to the specification, whereas the computation of the system time, which is needed to evaluate the timing constraints, has no direct connection to the specification.

To describe the connection between specification and realization, the compiler annotates the code fragments with their direct connection to the specification. Consider the following example:

```
--* c @{a,b}
c_en <= a_en and b_en;
```

With this line, the `VHDL` monitor evaluates the activation condition of the output stream `output c : Int8 @a,b := a + b`. In the specification, the corresponding part is described by `@{a,b}`. To related these parts, we annotate the previous code line. In our prototype, each code fragment of the compiled `VHDL` code is annotated with the corresponding part in the specification. For specification specific annotation, the comments start with `--*`, whereas general comments start with `--`. With these annotations, we visualize the relation between the code fragments and their corresponding encoding part in the specification, and document the code fragments. Additionally, we can understand the `VHDL` code based on the annotations without knowing the implementation details.

4.1. Entity Structure

In our prototype, we compile the different entities from templates to realize the specification with a hardware-based monitor. The different templates contain variables surrounded by the brackets `{{ }}`. These variables are placeholders that need to be filled by the compiler because they depend on the specification. The realization of the stream expression `a + 3` is an example of such a code fragment. At compile time, we receive the stream expression and build its realization, i.e. the placeholder for the stream expression realization `{{expr}}` is replaced by:

```
--* temp_0 := a
temp_0 := a_0;
--* temp_1 := b
temp_1 := b_0;
--* temp_2 := a +b;
temp_2 := temp_0 + temp_1;
updt := temp_2;
```

```
architecture structural of name is
  -- Component Declaration
    ...
begin
  -- Component Instantiation
    ...
end structural;
```

Figure 4.2.: Representation of the general structure of structural entities. Structural entities realize the connection between the different elements. Therefore, the architecture of these entities first declares the elements describing their input and output signals. Afterward, instances of the elements are established by connecting the input and output signals.

Structural Architectures

In our compilation, we differentiate between entities with a *structural*, *behavioral*, and *mixed* architecture. Figure 4.2 represents the general architecture of structural entities. They combine different elements of the monitor and consist of a component declaration phase specifying the input and output signals and a component instantiation phase mapping the input and output signals to concrete signals.

Behavioral Architectures

The general construction of a behavioral architectures is illustrated in Figure 4.3. Following the declaration, behavioral entities implement a new process to realize the logic of the monitor. Processes in our implementation start with an if-condition to separate the reset phase and the logic phase. The *reset phase* initializes all components to their initial state, e.g. resetting the monitor time to zero, and assinging the statical arrays to their statically known values, e.g. the deadline array from Section 3.1.4 is assigned to the global schedule for periodic streams. The *logic phase* of the behavioral entities is started by a rising edge of the clock signal to synchronize the parallel execution of the several entities. This phase realizes the logic for each component from Section 3.2.3, e.g. adding up the clock frequency to compute the system time.

→Sec. 3.1, P. 24

→Sec. 3.2, P. 35

Mixed Architectures

The last kind of entities are mixed architecture entities. These are a combination of structural and behavioral architectures and contain the declaration and instantiation of different components as well as a process to implement new functionality for the monitor.

4.2. Realization of Stream Types

The backend compiles every type to a corresponding numeric VHDL type from the ieee library. Table 4.4 presents all supported stream types and their corresponding VHDL types.

In comparison to the StreamLab interpreter, our prototype uses a fixpoint interpretation instead of a floating point representation. Arithmetic operators with fixpoint

```
1  architecture behavioral of name is
2    -- Internal Signal Declarations
3    signal register_1 : type_1;
4      ...
5    signal register_n : type_n;
6  begin
7    process(clk, rst) begin
8      if rst = '1' then
9        -- Reset Phase
10         ...
11     elsif rising_edge(clk) then
12       -- Logic Phase
13         ...
14     end if;
15   end process;
16   output_1 <= register_i;
17     ...
18   output_m <= register_j;
19 end behavioral;
```

Figure 4.3.: General structure of behavioral entities. Behavioral entities realize the logic of the monitor by implementing a process. The parameters, which have to be input signals of the entity, define the condition when to calculate new values. The prototype separates the process into two phases, the reset and the logic phase. VHDL uses internal signals to store values over several cycles. After the process implementation, the specified code always maps the output signals to internal signals.

Stream Value Type	VHDL Numeric Type
Bool	std_logic
Int8, Int16, Int32, Int64	signed(u downto 0), $u \in \{8, 16, 32, 64\}$
UInt8, UInt16, UInt32, UInt64	unsigned(u downto 0), $u \in \{8, 16, 32, 64\}$
Float16, Float32, Float64	sfixed(u downto l), $(u, l) \in \{(4, -11), (8, -23), (11, -52)\}$

Table 4.4.: Stream value types and their corresponding VHDL types.

representations use the same operators as integer types, which are highly optimized. The fixpoint representation in `VHDL` requires the bit range as parameter. In comparison to an integer type, the lower bound of the range is a negative number instead of zero. This representation uses the bits with a non-negative position for the integer part of the input. The bits with a negative position define the bit range for the fractional part. To interpret a bit vector $b_{n+m} \dots b_0$ as a real number of range `n downto -m`, we compute

$$-1 \cdot b_{m+n} \cdot 2^n + \sum_{0 \le i < n} b_{i+m} \cdot 2^i + \sum_{0 \le j < m} b_j \cdot 2^{j-m}$$

Based on this definition, we interpret the bit vector 011101 with range `2 downto -3` as:

$$\begin{aligned} 011.101 &= -1 \cdot (0 \cdot 2^2) + (1 \cdot 2^1) + (1 \cdot 2^0) + (1 \cdot 2^{-1}) + (0 \cdot 2^{-2}) + (1 \cdot 2^{-3}) \\ &= 2 + 1 + \frac{1}{2} + \frac{1}{8} \\ &= 3.625 \end{aligned}$$

4.3. High-Level Controller

→ Sec. 3.2, P. 35

The realization of the HLC follows the basic structure from Section 3.2.3. The implementation of the HLC differentiates between periodic and event-based streams, utilizing the same components. The periodic part computes the current system time of the monitor with the TimeSelect and passes it to the Scheduler. Then, the Scheduler decides if a deadline of a periodic stream arrives, and notifies the HLQInterface. The event-based side receives the incoming events with the ExtInterface and passes the parsed input to the EventDelay component. The realization is extended with a new component — called CheckNewInput— to decide if the current input is a new incoming event. This component waits on a rising edge of the `new_input` signal, such that the HLC pushes the input values to the Queue only if new values are received. Therefore, the output of this component is an input of the EventDelay, which uses this signal as the new incoming event flag. The EventDelay entity also receives the incoming event from the ExtInterface and the system time from the TimeSelect component. It annotates the input values with the current timestamp and delays the incoming event. The HLQInterface receives the time-annotated event at the same clock cycle with the time-annotated deadline from the Scheduler. Then, the HLQInterface decides which streams are affected by the deadline or the event. Additionally, the entity pushes the enable signals and data values to the Queue.

4.3.1. TimeSelect

The process of the TimeSelect entity computes the system time of the monitor in nano seconds based on the system clock. For this computation, we use a behavioral architecture which iteratively increases the time register in the logic phase. This computation

is independent of the specification, so the compiler does not relate the entity with the specification directly. For further details, Appendix A.2.1 presents the process implementation of the TimeSelect template.

→App. A.2, P. 116

4.3.2. The Scheduler Template

The Scheduler, presented in Figure 4.5, receives the system time from the TimeSelect entity as input signals and determines if a deadline of a periodic stream is due. For this component, only the periodic streams of the specification are relevant, more concretely their frequencies. These frequencies are needed to compute the hyper-period and the offset array, as presented in Section 3.1.4. To relate the offset array from the Scheduler with the frequencies in the specification, the implementation starts with an annotation containing the periodic streams of the specification. Additionally, the annotation visualizes the resulting hyper-period, and the offset array. To implement the logic, the entity uses a behavioral architecture. In the reset phase, the implementation initializes the array `offset_per_deadline` realizing the offset array. For these initializations, the compiler replaces the placeholder `{{dls}}` with the `VHDL` code for these assignments. To document them, we annotate this code fragment with the offset array which is the corresponding part in the specification. In the logic phase, we determine the arrival of a deadline by comparing the current system time against the timestamp of the next deadline. In case of a deadline, we update the time of the next deadline with the static offset array, update the ID of the deadline, and notify the HLQInterface. By using the static offset array, this code fragment is independent of the concrete specification.

→Sec. 3.1, P. 24

In comparison to Section 3.2.3, the output of the Scheduler does not contain the ID of the deadline. Therefore, the HLQInterface has to track this information by itself. This design decision reduces the number of bits sent by the Scheduler, but introduces a new register for the HLQInterface. However, it does not change the semantics of the realization.

→Sec. 3.2, P. 35

Example 4.3.1 (Scheduler Realization)**.** Consider the following specification:

```
input a : Int8
input b : Float16
output c : Float16 @2Hz := b.hold().defaults(to: 0.0) + 2.0
output d : Int8 @5Hz := a.aggregate(over: 2s, using: ∑)
output e : Int8 := a.offset(by:-1).defaults(to:0) - 3
output f : Int8 := e + b
```

To realize the timing schedule, the Scheduler uses the frequencies of the periodic streams `c` and `d`. From the frequencies, we build the hyper-period and the offset array, describing the time difference between the deadlines. To relate the schedule with the specification, the compiler produces the following annotation on top of the file:

```
--* Periodic Streams in Specification:
--* - c @2Hz
--* - d @5Hz
--* Hyper-Period: 1s
--* Offset Array in Seconds:
```

```
--* Periodic Streams in Specification: {{periodic_streams}}
--* Hyper Period: {{hyper_period}}
--* Offset Array in Seconds: {{offset_array}}

-- Internal Signal Declarations
signal time_of_next_deadline : unsigned(63 downto 0);
signal offset_per_deadline
  : unsigned64_array(({{size_dls}}-1) downto 0);
signal last_deadline_id : integer;

begin

process(clk, rst) begin
  if (rst = '1') then
    -- Reset Phase
    time_of_next_deadline <= to_unsigned({{init_dl}}, 64);
    last_deadline_id <= 0;
    time_last_deadline <= (others => '0');
    hit_deadline <= '0';
    -- Initialization of the Deadline Offset Array
    --* Offset Array in Seconds: {{offset_array}}
    {{dls}}
  elsif (rising_edge(clk)) then
    -- Logic Phase: Decision, if Arrival of a Deadline
    if (time_in >= time_of_next_deadline) then
      -- Deadline is reached
      time_of_next_deadline <= time_of_next_deadline
        + offset_per_deadline(last_deadline_id);
      last_deadline_id <= (last_deadline_id + 1) mod {{size_dls}};
      hit_deadline <= '1';
      time_last_deadline <= time_of_next_deadline;
    else
      -- No deadline is reached
      hit_deadline <= '0';
    end if;
  end if;
end process;
```

Figure 4.5.: Template for the process in the Scheduler entity. The Scheduler realizes the detection of deadlines for periodic streams. For these computations, the compiler stores the timestamp with the arrival of the next deadline and compares this value with the current system time. The prototype uses an offset array compiled out of the input specification to compute these timestamps.

```
--* || 0.2 | 0.1 | 0.1 | 0.2 | 0.2 | 0.2 ||
```

In the reset phase, we initialize the array with the offsets computed from the frequencies. To visualize the entries, we annotate the initialization with:

```
--* Initialization of the Deadline Offset Array
--* Offset Array in Seconds:
--* || 0.2 | 0.1 | 0.1 | 0.2 | 0.2 | 0.2 ||
offset_per_deadline(0) <= to_unsigned(200000000, offset_per_deadline(0)'length);
offset_per_deadline(1) <= to_unsigned(100000000, offset_per_deadline(1)'length);
offset_per_deadline(2) <= to_unsigned(100000000, offset_per_deadline(2)'length);
offset_per_deadline(3) <= to_unsigned(200000000, offset_per_deadline(3)'length);
offset_per_deadline(4) <= to_unsigned(200000000, offset_per_deadline(4)'length);
offset_per_deadline(5) <= to_unsigned(200000000, offset_per_deadline(5)'length);
```

In the template, the place holder `{{offset_array}}` is replaced by the annotation and the `{{dls}}` placeholder by the initialization assignments. The annotation are reflects the offset array from the frequencies in the specification. We see that the code fragment and the annotation are represented in the same way except for the time unit. The RTLola specification uses seconds and the realization nano seconds. △

4.3.3. CheckNewInput

When handling an incoming event, the monitor has to care about the freshness of the event and update the stream values only once per incoming event. In the realization, every signal can exclusively be set by the monitor or the monitored system. Consequently, the ExtInterface entity in the monitor cannot clear the `new_input` bit, set by the monitored system from Section 3.2. To solve this issue, the approach in the realization requires the monitored system to unset the `new_input` bit for at least one clock cycle before sending an event. This check is moved into a new component, the CheckNewInput entity, which implements this check in the logic phase. This logic is independent of the specification, such that the compiler performs no input specific annotations. For further details, Appendix A.2.2 shows the behavioral architecture of the template.

→Sec. 3.2, P. 32

→App. A.2, P. 117

4.3.4. ExtInterface

This entity receives from the monitored system the input events as a bit-representation and parses them to their VHDL numeric types. This information is encoded in the input stream declaration of the specification. Therefore, the compiler annotates the entity with these declarations on top of the file and additionally before each type-cast. This guarantees that the prototype compiles the correct casts by relating the stream value type with the VHDL numeric type. Figure 4.6 presents the different conversions from the bit-vector, received with the $s^{\downarrow}$`_in` signal, to the corresponding numeric type. Note that VHDL can infer the bit range for the signed and unsigned integer type. However, for the fixed-point conversion, we have to define the bit range explicit for determining the

Stream Value Type	VHDL Type Cast
Bool	$s^{\downarrow}$_out <= $s^{\downarrow}$_in
Int8, Int16, Int32, Int64	$s^{\downarrow}$_out <= signed($s^{\downarrow}$_in)
UInt8, UInt16, UInt32, UInt64	$s^{\downarrow}$_out <= unsigned($s^{\downarrow}$_in)
Float16, Float32,Float64	$s^{\downarrow}$_out <= sfixed($s^{\downarrow}$_in, u, l)
	$(u,l) \in \{(4,-11),(8,-23),(11,-52)\}$

Figure 4.6.: Stream value types and their corresponding type casts

→ App. A.2, P. 117

number of integer bits and fractional bits. Appendix A.2.3 presents the template if you are interested in the implemtation details.

Example 4.3.2 (ExtInterface Realization)**.** The specification from Example 4.3.1 contains the input streams a and b. To relate the type conversion in the ExtInterface with the specification, we annotate the entity with the input streams and their corresponding types. This results in the following annotation:

```
--* Input Stream and their Types in the Specification
--* - input a : Int8
--* - input b : Float16
```

Additionally, we annotate each conversion separately with the corresponding stream to relate the VHDL numeric type with the stream value type in the specification. For the specification, the compiler produces the following code fragment for the logic phase:

```
--* input a : Int8
a_parsed <= signed(a_data_in);
a_push_delayed <= a_push_in;
--* input b : Float16
b_parsed <= sfixed(b_data_in, 4, -11);
b_push_delayed <= b_push_in;
```

△

4.3.5. EventDelay

The EventDelay annotates the events from the ExtInterface with the current system time and forwards the events to the HLQInterface. To document this logic, we annotate the entity with all input streams, to check if the implementation covers each input stream. The implementation assigns in the logic phase the input signals to the corresponding output signals if the CheckNewInput identifies a new event.

```
1  process(clk, rst) begin
2    if (rst = '1') then
3      -- Reset Phase
4      time <= (others => '0');
5      push <= '0';
6      deadline_pos <= 0;
7      {{en_no_stream}}
8      clk_cycle_count <= 0;
9      -- Initialize Deadline Arrays
10     --* Deadline Array {{deadline_array}}
11     {{init_dl_arrays}}
12   elsif (rising_edge(clk)) then
13     clk_cycle_count <= (clk_cycle_count + 1) mod 4;
14     if next_deadline = '1' and clk_cycle_count = 0 then
15       -- Deadline Handling
16       push <= '1';
17       deadline_pos <= (deadline_pos + 1) mod {{deadline_len}};
18       time <= time_in_periodic;
19       {{deadline_handling}}
20     elsif new_input_in = '1' next_deadline = '1' and clk_cycle_count = 2 then
21       -- Event Handling
22       push <= '1';
23       time <= time_in_event;
24       {{event_handling}}
25     else
26       -- Enable No Stream
27       push <= '0';
28       {{en_no_stream}}
29     end if;
30   end if;
31 end process;
```

Figure 4.7.: Template for the process in the HLQInterface entity. This component decides on the one hand, which periodic output stream is affected by the arrival of a deadline. On the other hand, it also decides which input streams and event-based output streams are affected by an incoming event. For this detection, the logic phase is divided into a deadline handling case and an event handling case. The deadline handling case disables all event-based streams and uses a deadline array for periodic streams. These arrays are compiled out of the frequency and the hyper-period. In the event-based case, the compiler uses the enable input signals to build the activation condition for the input streams and the event-based output streams. The periodic streams are disabled.

4.3.6. HLQInterface

The HLQInterface decides which streams are affected with an incoming event or the arrival of a deadline. In the specification, the activation condition formulates this logic. For this reason, the entity starts with the following lines to relate the activation conditions of the specification with their realizations.

```
--* Streams and their Activation Conditions:
--* Input Streams: {{input_streams}}
--* Event-based Output Streams: {{event_output_streams}}
--* Periodic Output Streams: {{periodic_output_streams}}
--* Deadline Array: {{deadline_array}}
```

Dependent on the type of the stream, i.e. event-based or periodic, the realization of the activation condition is different. For this reason, these comments group the streams in input streams, event-based output streams and periodic output streams. Additionally, it assigns each stream to its activation condition defined in the specification or inferred by the StreamLab frontend. The realization to evaluate the activation condition of periodic streams uses the deadline array from Section 3.1.4, which encodes the schedule for periodic streams. To visualize this schedule, we annotate the component with the array from the specification. Additionally, the HLQInterface pushes the result of the activation condition with the timestamp and the event data to the Queue. The LLC then does not differentiate between periodic and event-based streams or the activation condition in general.

→Sec. 3.1, P. 24

Figure 4.7 shows the template with the behavioral architecture of the entity. Similar to the Scheduler, the component uses the hyper-period from the different frequencies in the specification to handle deadlines. The compiler builds for each periodic stream in the specification an array, which maps each deadline to a boolean value. This boolean value indicates if the stream has to be evaluated with the arrival of the current deadline. As the offset array in the Scheduler, this array is initialized in the reset phase. To visualize the entires, this code fragment is additionally annotated with the deadline array from the specification. The template separates the logic phase into three parts: *deadline handling*, *event handling*, and *no stream enabling*. These cases realize the different evaluation of the activation condition by assigning the enable signals of the streams to their activation conditions. The compiler annotates each assignment with the activation condition encoded in the specification to document them. In the deadline case, periodic streams are assigned to the current position in the deadline array, encoding their activation condition. In this case, the incoming event is not handled and all event-based streams are disabled. For the event-based part, the enable signal of input streams are mapped to the corresponding bits in the incoming events, encoding their activation conditions. For event-based output streams, the compiler inserts an expression, which represents the conjunction for the evaluation of the activation condition. All periodic streams are disabled in this case. In the else case, the HLQInterface does not receive an event or a deadline and disables all streams.

Deadline Handling

Event Handling

Depending on sending an event or a deadline, the interface uses the time of the Scheduler or the EventDelay as timestamp and pushes the time to the queue. Since

the HLQInterface has to send an incoming event and a deadline in one cycle of the HLC, the HLQInterface is assumed to tick four times as fast. In the first cycle, the HLQInterface handles the arrival of a deadline. In the second cycle, the HLQInterface has to unset the push signal of the queue. The third cycle then pushes the incoming event to the queue, and the last one unsets the push signal again. Without disabling the push signal, the clock cycles would not receive a rising edge of the push signal, that is needed by the Queue to push the data.

Example 4.3.3 (Scheduler Realization)**.** Consider again the specification in Example 4.3.1, with the input streams a and b, the periodic output streams c and d, as well as the event-based output streams e and f. Depending on the type of the stream, i.e. event-based or periodic, the realization of the activation condition is different. For this reason, the compiler groups the stream and produces the following annotation after the entity declaration:

→Ex. 4.3.1, P. 59

```
--* Streams and their Activation Conditions:
--* Input Streams:
--* - a @{a}
--* - b @{b}
--* Event-based Output Streams:
--* - e @{a}
--* - f @{a,b}
--* Periodic Output Streams:
--* - c @2Hz
--* - d @5Hz
--* Deadline Array:
--* || d | c | d | c | d | c | d | c, d ||
```

The evaluation of the activation condition for periodic streams is realized with the global schedule encoded by the deadline array. To document the initialization of this schedule in the reset phase, the compiler visualizes the deadline array of the specification with an annotation. This results in the following initialization:

```
--* Initialization of the Deadline Arrays
--* Deadline Array:
--* || d | d | c | d | d | c, d |
c_deadline_array(0) <= '0';
c_deadline_array(1) <= '0';
c_deadline_array(2) <= '1';
c_deadline_array(3) <= '0';
c_deadline_array(4) <= '0';
c_deadline_array(5) <= '1';

d_deadline_array(0) <= '1';
d_deadline_array(1) <= '1';
d_deadline_array(2) <= '0';
d_deadline_array(3) <= '1';
d_deadline_array(4) <= '1';
d_deadline_array(5) <= '1';
```

We see that the code fragment initializes the same array as the annotation, with the difference that the realization uses a bit array to indicate if a periodic stream is part of the current deadline. In the deadline handling phase, we then assign the periodic streams to the current value in the bit array encoding the schedule and disable all event-based streams. To relate and visualize these assignments with the corresponding part in the specification, we annotate them with the activation condition. This results in the following code fragment for the deadline handling.

```
--* a @{a}
a_en_push <= '0';
--* b @{b}
b_en_push <= '0';
--* c @2Hz
c_en_push <= c_deadline_array(deadline_pos);
--* d @5Hz
d_en_push <= d_deadline_array(deadline_pos);
--* e @{a}
e_en_push <= '0';
--* f @{a, b}
f_en_push <= '0';
```

We see that all periodic streams are assigned to the schedule whereas the event-based streams are disabled. For the event-handling, the prototype assigns the input streams to the enable signal from the incoming events. The event-based output signals are assigned to the encoding of their activation condition, and periodic streams are disabled. As for the periodic handling, the compiler annotates each assignment. This results in the following code fragment for the event-based handling:

```
--* a @{a}
a_en_push <= a_en;
--* b @{b}
b_en_push <= b_en;
--* c @2Hz
c_en_push <= '0';
--* d @5Hz
d_en_push <= '0';
--* e @{a}
e_en_push <= a_en;
--* f @{a, b}
f_en_push <= a_en and b_en;
```

△

4.3.7. High-level Controller

The top element of the HLC is an entity with a mixed architecture. With the previously described changes, the entity contains the single components of the HLC and maps their input and output signals. Additionally, it contains a new process which slows down the input clock for the Scheduler, the ExtInterface, the CheckNewInput, and the

EventDelay. Appendix A.2.4 represents the process template for the HLC, if you are interested in the implementation details. →App. A.2, P. 118

4.4. Low-Level Controller

The structure of the Low-Level Controller (LLC) in the realization is the same as in Section 3.2. The LLC contains LLQInterface with the state machine to pop the values from the queue. Additionally, the LLC contains a component which provides a single entity for each stream and sliding window in the specification. This entity is called the Evaluator in the realization and also implements the state machine in the EvalController with the mixed architecture. As in Section 3.2, the Evaluator in the realization receives the information which streams have to be updated with the current input values. The single timing issue of the Evaluator is the evaluation order of the specification. The following sections describe the LLC templates, starting with the compilation of input streams. →Sec. 3.2, P. 32

4.4.1. Input Streams

The prototype compiles a new component for each input stream $s^{\downarrow}$ in the specification. This entity receives the input data from the HLQInterface and provides the accessed offset values to the output streams. The specification encodes the information of the value type in the input stream declaration. The storage requirement for a stream is inferred from the dependency graph, as shown in Section 3.1.4. For valid RTLola specifications, the storage requirement for each stream is finite and statically known. Therefore during the compilation, the compiler builds an array storing all accessed values. To relate this array, i.e. the numeric type and the size, with the specification, the entity starts with: →Sec. 3.1, P. 24

```
--* Input Stream: {{input_stream}}
--* Input Dependencies: {{input_dependencies_in_dg}}
--* Storage Requirement: {{storage_requirement}}
```

The first line annotates the entity with the input stream declaration to relate the value type of the stream with the VHDL numeric type for each entry in the array. The second line annotates the realization with the input edges for this stream in the dependency graph, followed by the resulting storage requirement. This relates the specification with the size of the array to guarantee that the implementation covers all lookups.

The implementation of the process in input stream entities is nearly independent of the value type and the storage requirement of the current input stream. Figure 4.8 presents the process in the behavioral architecture. In the reset phase, the process assigns each signal to its default value and prepares the entity for the next input. Additionally, it assigns all lookups to invalid such that default expressions take the default value, because the stream lookup fails. The logic phase performs the update when receiving a new event. In this case, the entity shifts the data array by one and adds the current event value in the first position. Additionally, the phase marks the current input as valid. Because of the parallel execution of hardware, this shift produces no overhead

Update Case

```
1  architecture behavioral of {{name}}_entity is
2    -- Internal Signal Declarations
3    signal data : {{array_ty}};
4    signal data_valid : bit_array({{array_size}} downto 0);
5    signal done : std_logic;
6  begin
7    process (clk, rst) begin
8      if (rst='1') then
9        -- Reset Phase
10       done <= '0';
11       data(data'high downto 0) <= {{array_default}};
12       d_valid(d_valid'high downto 0) <= (others => '0');
13     elsif (rising_edge(clk)) then
14       -- Logic Phase
15       if (upd='1' and done = '0') then
16         -- Register Update
17         data <= data(data'high-1 downto 0) & d_in;
18         d_valid <= d_valid(d_valid'high-1 downto 0) & '1';
19         done <= '1';
20       elsif (upd='0') then
21         -- Reset done Signals
22         done <= '0';
23       end if;
24     end if;
25   end process;
26   -- Mapping: Register to Output Wires
27   d_out <= data;
28   d_valid_out <= d_valid;
29   done_out <= done;
30 end behavioral;
```

Figure 4.8.: Template for input stream entities. The LLC builds for each input stream in the specification a new entity out of the template. The logic phase of this entity updates the stream values with the incoming events. The output of the entity is a data array of size n, with the last n stream values. The compiler determines the size of the array from the dependency graph.

compared to a ring buffer implementation. After the Evaluator has updated all streams, the Evaluator disables all update signals, and the input stream entity prepares for the next event.

Example 4.4.1 (Input Stream Realization)**.** Consider the input stream a in the specification from Example 4.3.1. The compiler creates the following annotation and entity declaration for this input stream:

→Ex. 4.3.1, P. 59

```
--* Input Stream: input a : Int8
--* Input Dependencies:
--* Stream Lookups:
--* - e: -1
--* Window Lookups:
--* - d: (2s, sum)
--* Storage Requirement: 2
entity a_input_stream_entity is
  port (
    clk,rst : in std_logic;
    upd : in std_logic;
    d_in : in signed(7 downto 0);
    d_out : out signed8_array(1 downto 0);
    d_valid_out : out bit_array(1 downto 0);
    done_out : out std_logic
  );
end a_input_stream_entity;
```

The first annotation defines the corresponding input stream declaration in the specification for the input stream entity. The declaration contains the value type `Int8` of the stream which is realized with the `VHDL` numeric type `signed(7 downto 0)`. Therefore, the input signal `d_in` representing the value for an incoming event is declared with the `VHDL` numeric type. The output signal with the current stream values provided to the other streams is declared with the array type `signed8_array(1 downto 0)`. Each entry of the array is of the same numeric type `signed(7 downto 0)`, which is also documented by the stream declaration. The size of the array is related to the storage requirement of the stream. For this reason, the compiler annotates the input stream realization with the incoming edges in the dependency graph and the resulting storage requirement of the stream. △

4.4.2. Output Streams

Similar to input streams, we compile for each output stream $s^{\uparrow}$ in the specification, a single entity. Because the HLC computes the timing of the streams, the LLC does not differentiate between periodic streams and event-based streams anymore. In both cases, the compiler uses the same template which follows the same structure. An output stream entity evaluates the stream expression and provides the stream values to output streams. As for input streams, the value type of the stream is encoded with the output stream declaration, and the number of accessed values is inferred from the dependency graph. Output stream entities start with the following annotations:

```
process (clk, rst)
  -- Temporal Variables
  {{temporaries_declaration}}
  variable updt : {{ty}} := {{default_init}};
begin
  if (rst='1') then
    -- Reset Phase
    data(data'high downto 0) <= {{default_array}};
    d_valid(d_valid'high downto 0) <= (others => '0');
    pe_done <= '0';
    evaluate_fired <= '0';
  elsif (rising_edge(clk)) then
    -- Logic Phase
    if (pe = '1' and pe_done = '0') then
       -- Pseudo Evaluation
       data <= data(data'high-1 downto 0) & {{default_value}};
       d_valid <= d_valid(d_valid'high-1 downto 0) & '0';
       shift_done <= '1';
    elsif (eval = '1' and eval_done = '0') then
       -- Evaluation
       {{expr}}
       -- Register update
       data(0) <= updt;
       d_valid(0) <= '1';
       eval_done <= '1';
    elsif (pe = '0' and eval = '0') then
      -- Reset done Signals
      pe_done <= '0';
      eval_done <= '0';
    end if;
end process;
```

Figure 4.9.: Template for the process in output stream entities. The LLC builds for each output stream in the specification a new object out of the template. These entities realize the stream expressions and store the n stream values based on their memorization bound. The logic phase is separated into three phases: The first one performs the pseudo evaluation and shifts the stream values by one. The second phase updates the stream value by realizing the stream expression. The last step then prepares the entity by resetting the done signals.

```
--* Output Stream: {{output_stream}}
--* Input Dependencies: {{input_dependencies_in_dg}}
--* Storage Requirement: {{storage_requirement}}
--* Output Dependencies: {{output_dependencies_in_dg}}
```

The first three annotations are the same as the annotations in input stream entities. The first line relates the stream value type with the VHDL numeric type, and the second and third annotations relate the storage requirement of the specification with the size of the data array in the realization. The new line visualizes the outgoing dependencies in the dependency graph for the current output stream. These edges represent all lookups that are used in the evaluation of the stream expression. For the stream evaluation, the entity in the realization receives these values as input signals. The annotation contains the offset value as well as the corresponding value type and documents the input signals for the output stream entity. Note that sliding window lookups are outsourced to a separate entity as in Section 3.2 and this lookup value is also received as an input signal. →Sec. 3.2, P. 32

The template for the process realization in the behavioral architecture is represented in Figure 4.9. As for input streams, the reset phase initializes the signals with their default values, declares all output values as invalid, and prepares the process for the next input. The logic phase in output stream entities is separated in three cases. The first one is the pseudo evaluation, as described in Section 3.1.2. The realization for this phase is similar to the update case in input stream entities. The implementation shifts the data array, which evicts the last entry and fills the first position with a pseudo value, which is never accessed because of the evaluation order. The second case is the evaluation of the stream. The backend compiles out of the stream expression VHDL code with the same behavior. During this expression computation, the realization uses variable assignments instead of signal assignments, which are evaluated sequentially. This increases the depth of the resulting circuit and slows done the clock signal. Because of this sequential execution, we can evaluate a stream expression in one clock cycle and do not have to separate the computation in several cycles. The compiler uses the intermediate representation of a stream expression and assigns each subexpression to a temporary variable. To relate these assignments with the stream expression in the specification, we annotate each temporary variable, with the corresponding subexpression in the specification. If you are interested in the realization of the stream expressions, Appendix A.3 describes in detail how to compile the VHDL code from the stream expression in the specification. The last line of this compiled code fragment assigns the latest temporary variable containing the full expression to the updt variable. The last case prepares the stream entity for the next event, similar to input stream entities.

Pseudo Evaluation

→Sec. 3.1, P. 14

Evaluation

→App. A.3, P. 119

Reset Done Signals

Example 4.4.2 (Output Stream Realization)**.** Consider the output stream e in the specification from Example 4.3.1. The compiler compiles the following annotation and entity declaration for this output stream: →Ex. 4.3.1, P. 59

```
--* Output Stream: output e : Float16 := a.offset(by: -1).defaults(to:0) -3
--* Input Dependencies:
--* Stream Lookups:
--* - f: 0
```

```
--* Storage Requirement: 1
--* Output Dependencies:
--* - a of Type Int8 : -1
entity e_output_stream_entity is
  port (
    clk,rst : in std_logic;
    pe, eval : in std_logic;
    a_neg1 : in signed(7 downto 0);
    a_valid_neg1 : in std_logic;
    d_out : out signed8_array(0 downto 0);
    d_valid_out : out bit_array(0 downto 0);
    pe_done_out : out std_logic;
    eval_done_out : out std_logic;
  );
end e_output_stream_entity;
```

The first annotation defines the corresponding stream declaration in the specification. This relates the value type of the stream with the used VHDL type. The second and third annotations describe the storage requirement for the stream, which is realized with the size of the data array. These are the same annotations as for input streams. The last annotation describes the outgoing edges in the dependency graph. In our example, the stream expression `a.offset(by: -1).defaults(to:0) - 3` accesses the previous value of the input stream `a`. This value as well as a bit indicating if the lookup is valid are represented with the input signals `a_neg1` and `a_neg1_valid`. To document the type of these signals, we annotate each lookup with the corresponding VHDL type.

The prototype compiles the realization of stream expression and fills this code fragment at the `{{expr}}` placeholder. In our example, the stream expression is realized as:

```
--* temp_0 := a.offset(by:-1)
temp_0 := a_neg1;
--* temp_1 := 0
temp_1 := 1;
--* temp_2 := a.offset(by: -1).defaults(to: 0)
temp_2 := sel(temp_0, temp_1, a_valid_neg1);
--* temp_3 := 3
temp_3 := 3;
--* temp_4 := a.offset(by:-1).defaults(to: 0) + 3
temp_4 := temp_2 + temp_3

updt := temp_4
```

We assign each subexpression to a temporary variable and document these expressions with the corresponding part in the specification. △

4.4.3. Sliding Windows

A sliding window expression aggregates with a function γ over all values of a stream s inside a specific period. For an efficient evaluation of sliding windows, we use the

bucket approach from Section 3.1.4. This evaluation model restricts the aggregation function γ to list homomorphism. Recap, a list homomorphism $\gamma : A^* \to B$ consists of an unary map $map_\gamma : A \to T$ and a finalization $fin_\gamma : T \to B$ function, an associative binary reduction $\circledast_\gamma : T \times T \to T$ and a neutral element ε_γ. It allows splitting the computation of a sliding window into sub-aggregations, represented by the buckets. Because a sliding window expression appears only in a periodic stream with a fixed frequency, the compiler computes the number of buckets out of the frequency of the source stream and the duration of the window.

→Sec. 3.1, P. 24

A sliding window entity in the realization, pre-aggregates the entries of the buckets for each incoming event and return the result if requested. In the specification, this is all encoded with the single sliding window expression. To relate a sliding window entity in the realization with the corresponding part in the specification we annotate sliding windows with:

```
--* Sliding Window: {{sliding_window}}
--* Source Stream: {{source_stream}}
--* Number of Buckets: {{bucket_size}}
--* Time per Bucket: {{time_per_bucket}}
--* Input Type: {{type_of_aggregated_stream}}
--* Return Type: {{return_type}}
```

The first annotation contains the sliding window lookup in the specification, to check which sliding window is covered by the current entity. Additionally, the expression contains the duration of the window, which is needed to compute the number of buckets and the time per bucket, as presented in Section 3.1.4. The next annotation documents the source stream of the sliding window, i.e. the stream which contains the sliding window lookup. This annotation also presents the frequency of the source stream, which is used to compute the number of buckets and the duration of each bucket. The next two annotations then present these values computed from the specification. The number of buckets is related to the size of the array signal representing a bucket, and the time of each bucket is related to the signal with the timestamp of the next bucket creation. The last two annotations summarize the types of the aggregation function in the specification. The input type relates the specification with the numeric type of the input signal representing the current value of the target stream. i.e. the stream over which the sliding window aggregates. The return type documents the value type of the aggregation function, which is realized with the numeric type of the output signal.

Depending on the type of the stream and the used aggregation function, the prototype compiles a new entity from the template in Figure 4.10. The realization is dependent on the aggregation function, but each sliding window follows the same structure from Section 3.2. The realization uses an array where the size of the array equals the number of buckets $buckc_w$ to represent the different buckets. However, some aggregation functions use tuples as intermediate representation in the list homomorphism, which is not a supported type. To solve this issue, we split the array containing the buckets into several registers, one for each element in the intermediate representation. Because some aggregation functions return $\perp$ representing an invalid value for the sliding window, the

→Sec. 3.2, P. 32

```
process (clk, rst) begin
  if (rst='1') then
    -- Reset Phase
    evict_done <= '0';
    upd_done <= '0';
    request_done <= '0';
    data <= (others => '0');
    d_valid <= {{valid_upd}};
    last_ts_before_upd <= (others => '0');
    -- Reset Buckets
    data_valid_buckets(data_valid_buckets'high downto 0)
      <= (others => {{valid_upd}});
    {{set_sw_buckets_to_default_values}}
  elsif (rising_edge(clk)) then
    -- Logic Phase
    if (evict='1' and evict_done = '0') then
      -- Evict Case: New Timestamp
      if (time_in > last_ts_before_upd) then
        -- Update Timestamp
        last_ts_before_upd <= last_ts_before_upd + {{time_per_bucket}};
        -- Create New Buckets and Shift Bucket Array
        data_valid_buckets
          <= data_valid_buckets(data_valid_buckets'high-1 downto 0)
             & {{valid_upd}};
        {{create_new_and_shift_sw_buckets}}
      else
        -- No Timestamps Update
        evict_done = '1';
      end if;
    elsif (upd = '1' and upd_done = '0') then
      -- Update Case: Map New Input and Update Last Buckets Entry
      {{map_and_update_last_sw_bucket}}
      upd_done <= '1';
    elsif (request = '1' and request_done = '0') then
      -- Request Case: Finalize Buckets
      {{finalize_sw}}
      d_valid <= {{finalize_valid}};
      request_done <= '1';
    elsif (evict = '0' and upd='0' and request ='0') then
      -- Reset done register
      evict_done <= '0'; upd_done <= '0'; request_done <= '0';
    end if;
  end if;
end process;
```

Figure 4.10.: Template for the process in sliding windows entities. The logic phase of the sliding window realization is separated in the evict case, update case, and request case implementing the different behavior of windows.

code always contains a valid array, which determines for each bucket whether the bucket contains a value. Figure 4.10 represents the process of the template implementing the logic of a sliding window, separated into the reset and the logic phase.

The reset phase assigns the signals to their default values and prepares the entity for the next input as for output stream entities in Section 4.4.2. Based on the aggregation function, this phase also assigns each entry in the valid bucket to valid or invalid and assigns each bucket entry to the neutral element ε_γ. The logic phase is separated into the *evict case*, the *update case*, and the *request case*. The first phase checks if the buckets need to be updated with the current timestamp. In case of an eviction, the implementation updates the buckets by shifting them by one position. This shift deletes the last buffer, which contains the values that are not part of the window anymore and creates a new one initialized with the ε_γ value. Depending on the aggregation function, the buckets and the neutral element differ such that the compiler needs to compile the corresponding code fragments. A new timestamp could shift the buffer more than one position, so we end this phase only in case of no eviction. The second case of the logic phase performs the update of the "newest" bucket". For this update, the implementation lifts the incoming stream value to the representation of the list homomorphism by applying the map_γ function. Afterwards, it updates the bucket value v_{new} using the associative binary reduction function $\circledast_\gamma$ with the old value v_{old} and the lifted value as parameters. Formally, this approach is expressed as $v_{new} := v_{old} \circledast_\gamma map_\gamma(s^-)$. The last case covers the request of the sliding window, i.e. the situation that the periodic stream using the sliding window needs to be updated. As a consequence, this computation needs the current value on the window. The implementation concatenates the bucket values with the binary reduction function $\circledast_\gamma$ and applies the finalization function fin_γ on the result to compute the current sliding window value.

→Sec. 4.4, P. 69

Evict Case

Update Case

Request Case

Example 4.4.3 (Realization of Sliding Windows)**.** Consider the sliding window in the specification in Example 4.3.1: `a.aggregate(over: 2s, using: sum)`. The entity declaration and the corresponding annotation for this sliding window are presented with the following code fragment:

→Ex. 4.3.1, P. 59

```
--* Sliding Window: a.aggregate(over: 2s, using: sum)
--* Source Stream: d @5Hz
--* Number of Buckets: 10
--* Time per Bucket: 0.2 s
--* Input Type: Int8
--* Return Type: Int8
entity a_sum_sliding_window_entity is
    port (
        clk, rst : in std_logic;
        evict, upd, request : in std_logic;
        time_in : unsigned(63 downto 0);
        d_in : in signed(7 downto 0);
        d_out : out signed(7 downto 0);
        d_valid_out : out std_logic;
        evict_done_out : out std_logic,
```

```
        upd_done_out : out std_logic,
        request_done_out : out std_logic
end a_sum_sliding_window_entity;
```

First, we see the relationship between the value type for the target stream a and the numeric type of input signal d_in representing the input value in the update case. The return type of the aggregation function is reflected in the output signal d_out, providing the result of the request case. The number of buckets is reflected in the following internal signal declaration representing the buckets:

```
signal sum_buckets : signed8_array(9 downto 0);
```

Note that the realization starts with zero, whereas the specification starts counting with one. So the number of buckets is in both cases ten. The time per bucket annotation is reflected in the evict case of the logic phase, i.e. in the assignment for the timestamp with the next eviction:

```
last_ts_before_upd <= last_ts_before_upd + to_unsigned(200000000,
    last_ts_before_upd'length);
```

Note that the realization computes the timestamp in nano seconds whereas the specification uses seconds. △

Our prototype supports the aggregation functions counting, addition, averaging, and integration. In the following paragraphs, we describe the realization of counting and average by replacing the place holders. The interested reader may refer to Appendix A.3.1 for details about the addition and integration aggregation.

→ App. A.3, P. 125

Counting

The aggregation function counting over a stream s^- counts the number of computed stream values in s^-. Independent of the stream type T^- of stream s^-, the return type of this sliding window is of type UInt64. The following list homomorphism expresses the counting function mathematically: $count : A^* \rightarrow \mathbb{N}$

- $map_{count} : A \rightarrow \mathbb{N}$ with $map_{count}(x) = 1$
- $fin_{count} : \mathbb{N} \rightarrow \mathbb{N}$ with $fin_{count}(x) = x$
- $\circledast_{count} : \mathbb{N}^2 \rightarrow \mathbb{N}$ with $x_1 \circledast_{count} x_2 = x_1 + x_2$
- $\varepsilon_{count} = 0$

The compiler needs as internal register one bucket array, called count_buckets, which initializes each bucket entry to zero in the reset phase, to realize the counting aggregation function. Because the return value of this entity is always valid independent on how many stream values the sliding window receives, the compiler generates the code '1' for the place holder {{valid_upd}} and {{finalized_valid}}. To realize the update of the sliding window bucket, we perform the previously described shift and insert the neutral element 0. This approach results in the following realization for the {{update_sw_buckets}} placeholder:

```
count_buckets <= count_buckets(count_buckets'high-1 downto 0)
  & to_unsigned(0,count_buckets(0)'length);
```

To lift an incoming stream value and to update the last entry of the bucket, the prototype uses the code fragment:

```
count_buckets(0) <= count_buckets(0) + to_unsigned(1, count_buckets(0)'length);
```

The implementation adds up all bucket values to compute the final value of the counting window, realized by:

```
d <= count_bucket(0) + ... + count_bucket({{num_buckets}});
```

Averaging

One approach for the computation of the average value over some time, is to add up and counts all values and divides the sum by the number of events during the finalization. Another approach is the running average, which computes the average with each event but additionally counts the number of events to balance the current average value with the incoming event. Therefore, the list homomorphism uses a tuple for the intermediate representation to store the average and the count value. In the prototype, this approach is used for the fixed-point numeric type. The mathematical description for the averaging aggregation $avg : A^* \rightarrow A$ is:

- $map_{avg} : A \rightarrow (A, \mathbb{N})$ with $map_{avg}(x) = (x, 1)$
- $fin_{avg} : (A, \mathbb{N}) \rightarrow A$ with $fin_{avg}(x, c) = x$
- $\circledast_{avg} : (A, \mathbb{N})^2 \rightarrow A$ with $(x_1, c_1) \circledast_{avg} (x_2, c_2) = \left(\frac{c_1 \cdot x_1 + c_2 \cdot x_2}{c_1 + c_2}, c_1 + c_2\right)$
- $\varepsilon_{avg} = (0, 0)$

To realize the tuple used as the intermediate representation, the prototype uses two internal registers for the bucket approach, which initialize each bucket entry to zero. The `count_buckets` signal counts the number of received stream values per bucket and the `avg_buckets` signal stores the average value for each bucket. Similar to the counting aggregation function, the evict case shifts the buckets and inserts the default neutral element zero. In the update case, we replace the place holder `{{update_sw_buckets}}` with:

```
avg_buckets(0)
 <= resize((avg_buckets(0) * count_buckets(0) + d_in)
        / (count_buckets(0) + 1), u, l);
count_buckets(0) <= resize(count_buckets(0) + 1, u, l);
```

The request case computes the average of all buckets, by balancing their average value with the number of entries and dividing the result with the number of events. Therefore, the realization replaces `{{finalize_sw}}` with:

```
d <= resize(
  (avg_buckets(0) * count_buckets(0) + ... +
   avg_buckets({{num_buckets}}) * count_buckets({{num_buckets}})
  / (count_buckets(0) + ... + count_buckets({{num_buckets}}))) ,u, l);
```

Since the division by zero is not defined, which is possible, if no value was received inside the timed window, the entity notifies the stream expression to take the default value. Therefore, the return value for the `d_valid_out` is assigned to `'0'`, which is achieved by replacing the `{{valid_upd}}` place holder with `'0'`. This assignment defines each bucket entry to invalid. If an event is received, i.e. a rising edge of the `upd` signal, the valid entry for this bucket is assigned to one. Therefore, the compiler inserts the following assignment to the `{{map_and_update_last_sw_bucket}}` placeholder:

```
data_valid_buckets(0) <= '1';
```

In the request case, the compiler builds the disjunction of all `valid_buckets` entries for `{{finialize_valid}}`.

4.4.4. Evaluator

The Evaluator is the core element of the LLC. It receives the output from the Queue between the HLC and LLC and updates the different stream values. This entity receives flags from the LLQInterface expressing that the state machine is in the evaluation mode. This flag is set until the Evaluator notifies the state machine in the LLQInterface, that the evaluation cycle has finished. With this signal, the state machine can access a new value from the Queue.

To update the different stream values and sliding windows, and to coordinate these computations, the Evaluator needs a mixed architecture. It has the component instantiation and declaration for each stream and sliding window in the specification. The compiler annotates each declaration and instantiation with the corresponding stream and window. This annotation describes the category of the stream or sliding window, its name and its value type to relate the type declarations and signal mappings with the specification. The annotations for output streams additionally describe the stream expression to document the additional input signals for output streams. Because the LLC receives from the Queue, which streams need to be evaluated, the activation condition is not part of the documentation. The annotation relates the used signals with the corresponding stream in the specification. Additionally, it documents that each stream and sliding window is covered in the Evaluator.

Example 4.4.4 (Instantiation of Stream Entities in the Evaluator)**.** The instantiation for the input stream entity realizing the stream a in the specification from Example 4.3.1 is represented with the following code fragment:

→Ex. 4.3.1, P. 59

```
--* input a : Int8
a_input_entity_instance: a_input_entity
  port map (
    clk => clk,
```

```
    rst => rst,
    udp => a_udp,
    d_in => a_d_in,
    d_out(0) => a_d_0,
    d_out(1) => a_d_neg1,
    d_valid_out(0) => a_d_valid_0,
    d_valid_out(1) => a_d_valid_1,
    done_out => a_upd_done
  );
```

The instantiation assigns each input and output signal in the entity declaration to the corresponding signals in the Evaluator. The annotation relates the current instantiation a_input_entity_instance with the corresponding stream a in the specification. △

Besides the component declarations and instantiations, the entity has a new process that implements the state machine with the evaluation order from Section 3.1.2. The implementation of this state machine is done implicitly by assigning the stream enable signals to their enable signals, set by the HLC and their dependencies. The enable signals correspond to the result of the activation condition of the stream, such that the LLC can ignore them. This encoding is realized in the new process represented in Figure 4.11. In the reset phase, the stream-enable signals and sliding window-enable signals are disabled to prepare the streams and windows for the first input. The reset phase notifies the state machine in the LLQInterface that a new event can be received. The logic phase is separated into three parts, encoding the state machine. The first case covers the input stream update and the pseudo-evaluation phase of the state machine. In the state machine from Section 3.2, this code fragment implements the effect from the switch of the idle state to the first node. Additionally, this case has to cover the evict phase for sliding windows. To relate this realization with the evaluation order in the specification, we categorize the streams and windows in input streams, output streams, and sliding windows and annotate the corresponding code fragments with their stream names and sliding windows.

→Sec. 3.1, P. 10

→Sec. 3.2, P. 32

Example 4.4.5 (Realization Input Update, Pseudo Evaluation, and Sliding Window Evict Phase)**.** Consider the specification from Example 4.3.1. The compiler builds the following assignments and annotations for this specification:

→Ex. 4.3.1, P. 59

```
-- Pseudo Evaluation Phase, Input Stream Update Phase and Evict Phase
--* Input Streams:
--* - a
--* - b
a_upd <= a_en;
b_upd <= b_en;
--* Output Streams
--* - c
--* - d
--* - e
--* - f
c_pe <= c_en;
```

```
process(clk, rst) begin
  if rst = '1' then
    -- Reset Phase
    evaluator_done <= '1';
    upd_and_pe_done <= '0';
      {{disable_in_stream_upd_signals}}
      {{disable_pe_signals}}
      {{disable_eval_signals}}
      {{disable_evict_signals}}
      {{disable_sw_upd_signals}}
      {{disable_request_signals}}
  elsif rising_edge(clk) then
    -- Logic Phase
    if input_clk = '1' then
      if upd_and_pe_done = '0' then
        -- Pseudo Evaluation Phase, Input Stream Update Phase and Evict Phase
        --* {{input_streams_in_specifcation}}
        {{enable_in_stream_upd_signals}}
        --* {{output_streams_in_specifcation}}
        {{enable_pe_signals}}
        --* {{sliding_windows_in_specifcation}}
        {{enable_evict_signals}}
        upd_and_pe_done <= '1';
        evaluator_done <= '0';
      else
        -- Sliding Window Update Phase
        {{enable_sw_upd_signals}}
        -- Sliding Window Request Phase
        {{enable_request_signals}}
        --Eval phase
        {{enable_eval_signals}}
        --Done port assignment
        evaluator_done <= upd_and_pe_done and {{done_port_assignment}}
      end if;
    else
      upd_and_pe_done <= '0';
      {{disable_in_stream_upd_signals}}{{disbale_pe_signals}}
      {{disbale_eval_signals}}
      {{disable_evict_signals}}
      {{disable_sw_upd_signals}}
      {{disable_request_signals}}
    end if;
  end if;
end process;
```

Figure 4.11.: Template for the process in the Evaluator entity. This process realizes the state machine from Figure 3.11 coordinating the evaluation order of the stream and sliding windows evaluation.

```
d_pe <= d_en;
e_pe <= e_en;
f_pe <= f_en;
--* Sliding Windows:
--* - a.aggregate(over: 2s, using: sum)
a_sum_evict_sw <= '1';
```

The annotation relates the assignments with the corresponding part in the evaluation order in the specification. The implementation enables the update phase for input streams, the pseudo evaluation for output streams if the stream is affected by the current event. Additionally it unconditionally starts the evict case for sliding windows. △

The second case is separated in the eval phase, i.e. the enabling of output stream entities to evaluate their expression, the sliding window update phase, i.e. the enabling of sliding window entities to update the "newest" bucket, and the sliding window request phase, i.e. enabling of sliding window entities to request the current window. For the eval phase, we assign the enable signal of output stream entities to the outgoing dependencies in the DG. In the state machine in Section 3.2, these steps are performed by switching the state 2.n to 2.n+1. This corresponds to the computation of the evaluation layer for the specification. We annotate these assignments with the corresponding stream in the specification, and the outgoing dependencies in the DG, which access the current value of other output streams, i.e. synchronous lookups and sample & hold lookups, or access a stream with a sliding window. This relates the evaluation layer in the specification with their encoding in the implementation. The phases in the sliding window entities are enabled after the target stream is updated and before the source is evaluated, respectively. Therefore we annotate these assignments with the source and target stream in the dependency graph to relate these assignments with the evaluation order in the specification.

→Sec. 3.2, P. 32

Example 4.4.6 (Realization Eval Phase and Sliding Window Update & Request Phase)**.** Consider the specification in Example 4.3.1. The compiler builds the following assignments and annotations:

→Ex. 4.3.1, P. 59

```
-- Eval Phase
--* Evaluation Phase for Output Stream c is Influenced by no Lookup:
c_eval <= c_en;
--* Evaluation Phase for Output Stream d is Influenced by the Following Lookups:
--* - Window Lookups: a.aggregate(over: 2s, using: sum)
d_eval <= d_en and a_sum_request_done;
--* Evaluation Phase for Output Stream e is Influenced by no Lookup:
e_eval <= e_en;
--* Evaluation Phase for Output Stream e is Influenced by the Following Lookups:
--* - Synchronous lookups and Sample & hold Lookups: e
f_eval <= f_en and e_eval_done;
-- Sliding Window Update Phase
--* a.aggregate(over: 2s, using: sum) Targets a
a_sum_upd <= a_upd_done;
-- Sliding Window Request Phase
```

```
--* a.aggregate(over: 2s, using: sum) has Source d and Target a
a_sum_req <= d_en and (not a_en or a_upd_done);
```

The evaluation of the output stream `c` and `e` is in the second layer after the input streams. These streams do not access a current value of an output stream, or of a sliding window lookup, so after the input update, these streams are evaluated. The compiler annotates this information before the assignments and relates them with the evaluation order in the specification. Recap from Section 3.1.2 that the evaluation order is computed out of these lookups. The realization enables the evaluation of these streams if the current event affects the stream which corresponds to the second layer in the evaluation order. The output stream `d` aggregates over the input stream `a`. Because the sliding window lookup is outsourced into a new entity, the evaluation of the output stream waits until the result of the aggregation function is requested and computes then the value. This can delay the current evaluation layer by one clock cycle but does not change the semantics, because all affected streams are delayed recursively. This dependency in the specification is annotated by the compiler and reflected in the assignment. The output stream `f` has a synchronous lookup to the output stream `e`. Therefore the evaluation has to wait for the evaluation of `e`. In the specification, this is reflected by the different evaluation layers, computed from the dependency graph, described by the annotation. In the assignment, this is reflected by the conjunction. The annotation for the sliding window contains the source and target stream to reflect the behavior between specification and realization. The update of the sliding window is performed after the update of the target input stream `a`. The realization requests the sliding window if the source stream `d` is enabled. Because an event might update and request a stream, the request has to wait until a possible update is performed. This is in the assignment encoded with the implication. △

→Sec. 3.1, P. 10

Additionally, this case has to decide if the evaluator has performed all stream updates and evaluations for the current input. Therefore, we have to check if every stream that was enabled by the queue value has finished its computation. In this case, we notify the LLQInterface. In our backend, the implementation of the implication $s^{-}_en \rightarrow s^{-}_done$ realizes this logic for each stream s^{-}. This implication expresses that if a stream was enabled, the computation of this stream has finished.

4.4.5. Low-level Controller

The top entity of the LLC, the LLC entity itself, is an entity with a mixed architecture. It contains the Evaluator as a component and a process realizing the LLQInterface from Section 3.2. Details about the process realizing the state machine are presented in Appendix A.2.5. The input signals of the LLC are the output signals of the queue, and the output signals are the output signals of the Evaluator. The instantiation of the Evaluator maps the input and output signals to their respective signals, as described in the previous section.

→App. A.2, P. 118

4.5. Monitor

The top element of the monitor is an entity with a mixed architecture. The monitor contains as input signals:

- the different clocks
- an input bit indicating that new values are received
- for each input stream in the specification, a signal with the input data and a one bit signal, that indicates whether the current event contains a value for this input stream

The output signals for this entity are the system time of the monitor and the current values for each input and output stream. The architecture of the monitor contains the HLC, LLC, and Queue declaration, instantiation, and signals to connect them. It maps the clock signals to their respective components, the input signals of the monitor to the input signals of the HLC, the output signals of the HLC to the input signals of the queue, and the output signals of the queue to the input signals of the LLC. The output signals of the LLC are mapped to internal registers used by the new process. The process uses these registers to update the output values of the monitor only if the evaluation has finished. Therefore, it waits on a falling edge of the LLC that indicates that the computation is done.

Chapter 5

Integration

This chapter describes the integration of the prototype from Chapter 4 into a real-world domain. We integrated the hardware ZC702 Base Board from Xilinx[1], on which a compilation of our monitor is running, into an unmanned aerial vehicle (UAV). Therefore, we cooperate with the Unmanned Systems department at the German Aerospace Center (DLR) Institute of Flight Systems in Brunswick. The overall goal of this department is the development of easy and safe operations of unmanned aircraft. One step towards this goal is the project ALAADy (Automated Low Altitude Air Delivery). From the conceptual planning to an actual flying demonstrator, ALAADy considers low-altitude cargo air delivery of over 1t over hundreds of kilometer. When flying long distances, many issues may occur, e.g. data link loss. The UAV has to be autonomous to automatically react to such situations. Since proving the correctness of such a complex autonomous system is a hard task, monitoring can be used as the central safety mechanism. The presented prototype solves several safety concerns, e.g. static memory consumption or worst case runtime due to the inherent nature of the hardware solution. To validate the prototype for this domain, we integrate monitors running on a ZC702 Base Board into replays of pre-recorded flight. Next, we integrate and deploy the hardware-board into the UAV to identify the overall system impact. The following sections describe the mission that is monitored by our prototype, the integration in the re-play environment, the integration into the vehicle, and the general structure of the communication between the monitor and the UAV.

→Chap. 4, P. 53

5.1. The Mission

To implement a monitor as part of the ALAADy project, we need a monitor interface which gives the monitor an adequate amount of data to validate the properties. Therefore, the system needs to forward sensor values with realistic frequencies, which can

[1] https://www.xilinx.com/support/documentation/boards_and_kits/zc702_zvik/ug850-zc702-eval-bd.pdf

Figure 5.1.: Picture of the helicopter that is used for the flight test. This vehicle has a total mass of 150kg and a payload up to 85kg. The high payload allows besides the sensors needed for the flight execution also additional hardware like the board running the hardware monitor.

vary from sensor to sensor. Because such an interface is currently not provided by the ALAADy demonstrator, we integrated the monitor in a flight test [32, 33] using the ARTIS (Autonomous Rotorcraft Testbed for Intelligent Systems) testbed. This testbed consists of a software framework replaying pre-recorded flights and a fleet of UAVs. For this, technology and components for autonomous flights are deployed and evaluated. Part of this testbed is a helicopter, represented in Figure 5.1 — called superARTIS — with a payload up to 85kg. This payload is high enough to carry the sensor used for the optical navigation, and the hardware board running the monitor. The validation results, i.e. whether the prototype can hold up to data rates and is able to compute evaluation results in time, carry over from superARTIS to ALAADy since in both systems the same sensors are used.

The overall goal of the flight test is optical navigation based on landmarks. Figure 5.2 represents the flight test area with the different landmarks from a bird's-eye view. This area is prepared with individual landmarks of which the system knows their exact position. The landmarks are printed with individual QR-codes such that the on-board camera can recognize and differentiate them. Based on the ID of a recognized landmark and the distance to them, an algorithm then estimates the current position of the vehicle. Additionally, the setup records all the sensor data to validate the position as a post process. This logging procedure can be reused for our monitor interface by not only writing the data to file but also sending the data to the monitor.

Figure 5.3 represents the abstract structure of the system setup including the monitor. The system collects all data from the different sensors fixed on the payload with their

Figure 5.2.: A representation of the landmarks used for visual navigation in the test flight.

desired frequencies. These data packages are given to the state estimation algorithm but also to the `Logger`. The `Logger` writes the estimated position and the sensor values to files. Additionally, it directly sends them to the monitor via UDP. With this architecture, the system provides an interface for the monitor to receive raw data from sensors based on their frequencies and can compare the computed values from the monitor with the values from the system. Furthermore, because the system performs most of the communication without the monitor integration, the integration hardly changes the timing behavior. This property is desired, especially in real-world applications working with inputs from the environment, to guarantee the independence of the monitor and the monitored system and increases the confidence in the monitoring process. It is also essential for the development to test the system and the monitor independently of each other without changing the timing behavior.

5.2. Integration Setup

At first, we integrate the hardware-board in a simulation, replaying a pre-recorded flight with the same timing behavior. Figure 5.4a represents the replayer, which is a computer identical to the one used in the vehicle. In comparison to a real execution, the device receives the values from log files instead of real sensors. These files are generated from sensor readings during a flight and are stored to replay this specific flight. As during actual flights, the algorithm estimates the position from the different sensor values and

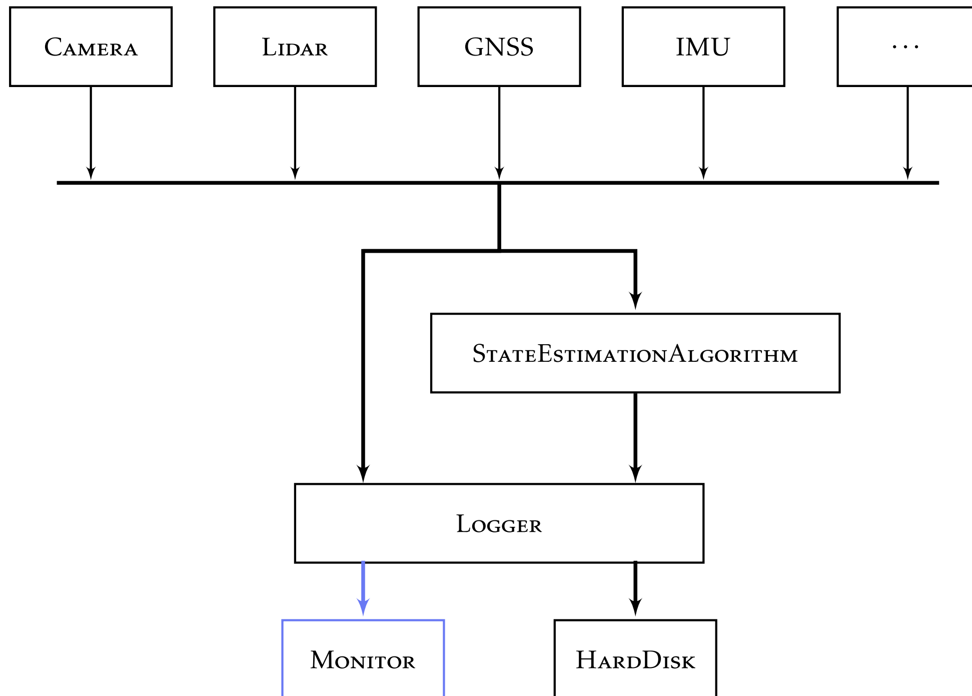
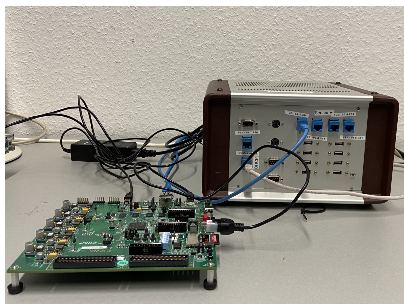
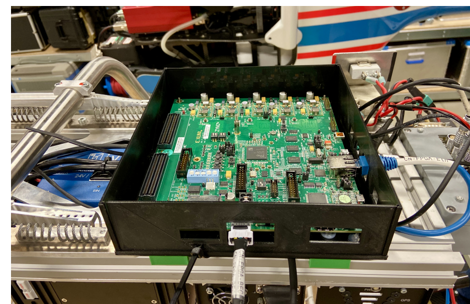


Figure 5.3.: A representation of the abstract structure of the system with the monitor. The state estimation algorithm receives values from the sensors in the payload and computes the current position of the vehicle. During the execution, the system logs the incoming and computed values with the Logger. Additionally, the Logger parses the received values to UDP packages to forward them to the monitor.



(a) Representation of the replayer that plays a prerecorded flight by reading sensor values out of log files generated during a flight and sending the values to the monitor.



(b) Representation the hardware board integrated in the payload. The board is fixed within a 3D-printed case on the payload and connected with the vehicle.

Figure 5.4.: Integration of the hardware board, where the monitor from Chapter 4 runs, in the simulation environment, and the payload.

sends the original values and the computed ones to the Logger. With this setup, the replayer plays a prerecorded flight with the same timing behavior as during the flight. For the replay, we establish an ethernet connection between the hardware board and the simulation device. Additionally, we adapt the execution to send the packages to the Logger as well as the hardware board, where the monitor is running.

We also integrate the monitor into the helicopter. The hardware board is located on the landing skids next to the sensor devices. We build a case out of a 3D-printer to protect the hardware from the environment. The case is also needed to attach the board on the landing skids, represented in Figure 5.4b. The case has an interface for an ethernet connection to bridge the board with the system. Additionally, it provides an interface for the power supply given by the vehicle and provides an interface for a SD Card. On this SD Card, the processor stores the file with the computed stream values and it also contains the boot image for the hardware board. In contrast to the simulation, the hardware board needs to boot from an SD card rather than with the JTAG[2] industry standard from an external device. Therefore, the board is implemented with a first-stage bootloader. When starting the device, the bootloader loads the bitstream onto the FPGA and configures the processing system.

5.3. BoardSetup

To integrate the hardware-based monitor compiled out of the prototype from Chapter 4 we use the Xilinx Zynq-7000 SoC ZC702 Evaluation Kit. This board is part of the Zynq-7000 SoC family that features a dual-core ARM Cortex-A9 processor integrated with an Artix-7 programmable logic and combines programmability of software and hardware. The used board provides a couple of interfaces, e.g. USB communication and ethernet, which allows integration with different interfaces. The architecture of the monitored system provides an interface with an ethernet connection sending UDP packages, which we used for the integration.

→Chap. 4, P. 53

The focus of this thesis is to show the practicability of a hardware-based monitor in a real-world application. We use the on-board processor for the communication with the monitored system and the programmable logic only for the monitor evaluation. Figure 5.5 is a representation of the general structure and control flow on the hardware board. The processor receives the packages sent by the vehicle or the simulation of a vehicle. Besides the IP-Header and the UDP-Header implementing the UDP protocol, the UDP-Data uses a separate header, represented in Figure 5.6. They start with an identification number (ID) from the device, sensor or algorithm which produces the values, the ID of the package, a timestamp, and an error code. Based on the ID of the package, the execution parses different input stream values from the data block. The packages are sent with different frequencies, which requires an asynchronous monitoring mechanism, that is supported by our monitoring approach. In the asynchronous setting, not all input streams have to contain a value in comparison to a synchronous setting,

[2] https://www.xilinx.com/support/documentation/boards_and_kits/zc702_zvik/ug850-zc702-eval-bd.pdf describes the different boot modes.

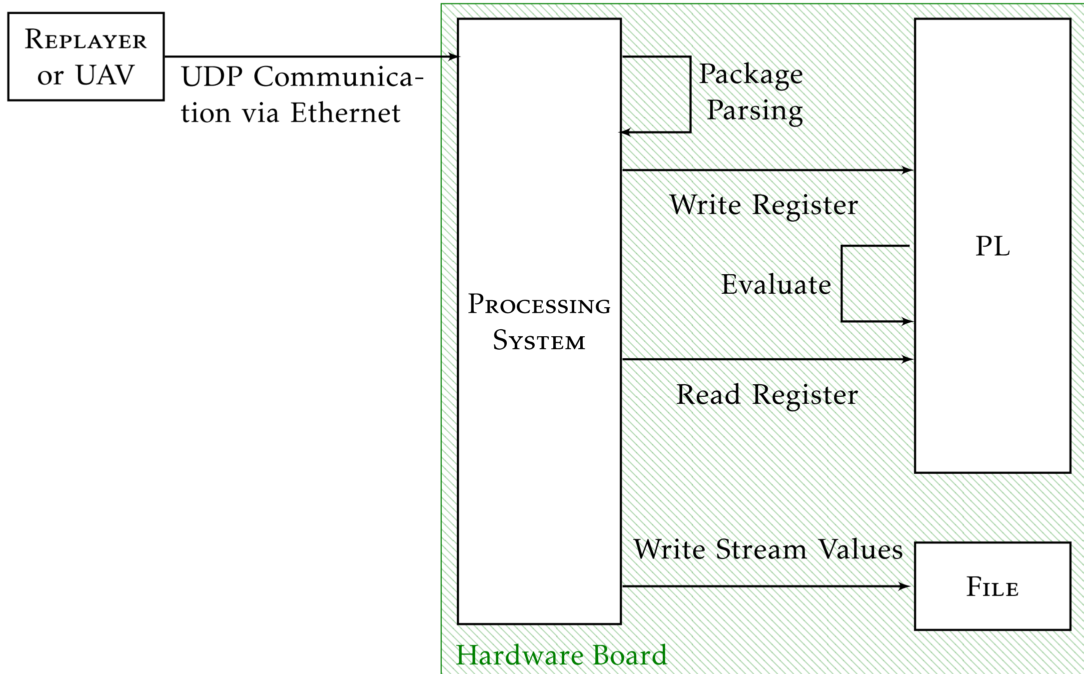


Figure 5.5.: Representation of the separation between software and hardware on the Xilinx Board. The processor receives the different asynchronous UDP packages via the Ethernet connection. Afterward, the parsed event is given to the programmable logic, synthesized with the hardware monitor. After the computation, the hardware gives the current stream values to the processor, which writes them to a file.

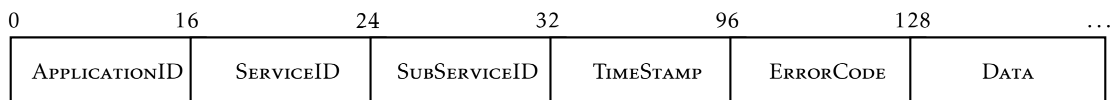


Figure 5.6.: Representation of the structure of the UDP-Data. The data fragment of each UDP package starts with the ID of the system, sensor or algorithm computing the values of the current package. Afterward, the `serviceID` and `subserviceID` stores the ID of the current package identifying how the following data has to be parsed. Then, each data package is annotated with a timestamp and an error code, followed by the current event, which is given to the monitor.

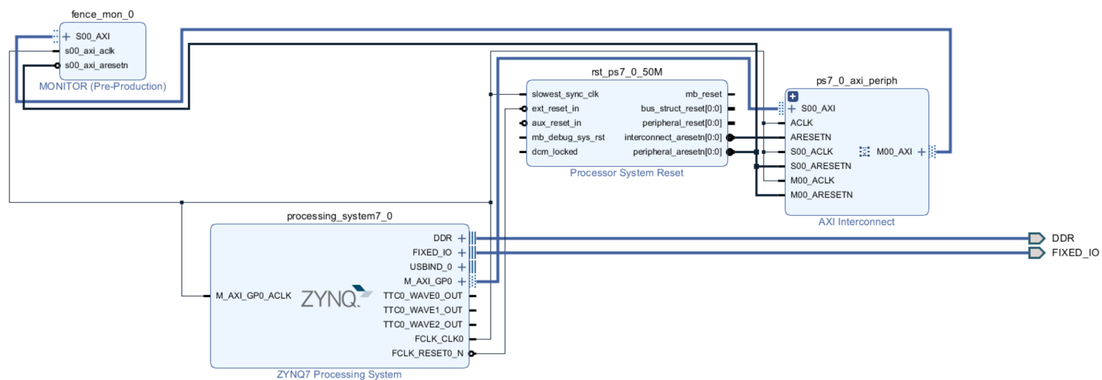


Figure 5.7.: Representation of the block design used for the hardware-based monitor integration. The AXI Interconnect component connects the processor, which handles the communication with the monitored system, with the hardware-based monitor. The monitor is integrated into a new AXI4 peripheral by assigning the registers to the input and output signals of the monitor.

where for each incoming event, each input stream always has to contain a value. After parsing the incoming package, the process forwards the information to the monitor running on the FPGA. Therefore, the processor transfers the different elements to their bit representation used on the monitor. This transformation is necessary especially for real-numbers, where the floating-point representation on the processor is transformed into the fix-point representation used for the programmable logic. The monitor then computes the different stream values and forwards them to the processor in the next step. In the last step, the processor parses the bit representation from the FPGA to C types and writes the current stream values into a file.

For the communication between the processing system and the programmable logic, we used the pre-defined AXI communication protocol[3]. This interface allows connecting multiple master devices with several slave devices by using read and write channels between the master and the slave. Figure 5.7 represents the block design of the implementation on the board. The Zynq processing system component, representing the processor, is the memory-mapped master device. The monitor, which is in our architecture the memory-mapped slave, is implemented as a new AXI4-peripheral. The AXI Interconnect component completes the communication between the monitor and the processing system. The AXI4-IP contains the `VHDL` monitor as a separate component and performs the communication part of the monitor. Therefore it implements the AXI protocol and maps the wires of the AXI Interconnect to their corresponding signals in the monitor. With this communication the master devices access variables from the slave devices and perform read or write operations.

[3] `https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf`

→Chap. 4, P. 53

The prototype from Chapter 4 compiles besides the VHDL file containing the monitor also VHDL code fragments needed for the AXI integration. Additionally, the compilation produces C files that implement the previously described approach. Further details about integration the VHDL files are presented in the Appendix A.4.

→App. A.4, P. 128

Chapter 6

Case Study

This chapter describes a case study for the prototype from Chapter 4 integrated into the simulation environment from Chapter 5. For this, we performed three experiments, geofencing, sensor validation, and cross validation.

→Chap. 4, P. 53
→Chap. 5, P. 85

Before we present the results of the case study, we first describe the corresponding specification in detail. For the geofence specification and the cross validation, we additionally describe the underlying model.

6.1. RTLola Specifications

6.1.1. Geofencing

In the first experiment we monitored a high-level command structure of our system, i.e. the operations of the vehicle. The example for the high-level control mechanism presented in this thesis is the position monitoring. During a flight, the vehicle has to avoid No-fly zones, e.g. nearby airports, and has to stay in the granted flight area. We realized the area in which the vehicle is allowed to fly with a polygon – called geofence. We check if the latitude and longitude values of the UAV are in the approved flight zone.

Underlying Model

To compute the crossing of a boarder in a geofence, we use vector arithmetic: In a first step, we describe each border in the fence as well as the flight path as a function. In a second step, we compute the intersections of these functions which correspond to potential intersections of the flight path with the geofence. Afterward, we check if the intersection is between the points describing the line of a fence and between the two samples describing the current vehicle function. Figure 6.1 gives an example, of such a border crossing. In our coordination system the x-axis represents the latitude (short: *lat*) and the y-axis the longitude (short: *lon*). The red line represents a geofence border between the points p_1 and p_2. The green line describes the monitored flight described

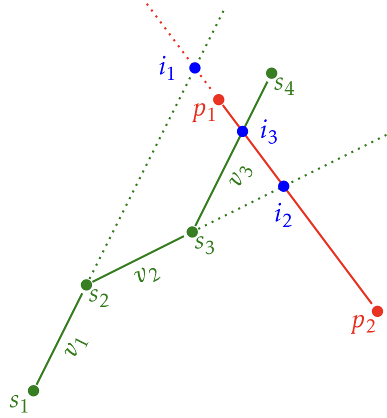


Figure 6.1.: Representation of the border crossing example. The red line represents the border of the geofence. The green line represents the flight path, constructed from the sample points s_1, s_2, s_3, and s_4.

by the sample s_1, s_2, s_3, and s_4. At first we define the function l_1, represented with the green dotted lines, based on the fence points p_1 and p_2, by:

$$
\begin{aligned}
lon_{p_1} &= m_{l_1} \cdot lat_{p_1} + b_{l_1} \\
lon_{p_2} &= m_{l_1} \cdot lat_{p_2} + b_{l_1} \\
&\Rightarrow m_{l_1} = \frac{lon_{p_2} - lon_{p_1}}{lat_{p_2} - lat_{p_1}} \\
&\Rightarrow b_{l_1} = lon_{p_2} - m_{l_1} \cdot lat_{p_2} \\
&\Rightarrow l_1(lat) = m_{l_1} \cdot lat + b_{l_1} = lon
\end{aligned}
$$

Next, we define the function v_n from two sample points s_n and s_{n+1} in the same way and compute the intersection between the function describing the geofence line l_1 and the function describing the current flight v_n by:

$$
\begin{aligned}
l_1(lat_{i_n}) &= v_n(lat_{i_n}) \\
m_{l_1} \cdot lat_{i_n} + b_{l_1} &= m_{v_n} \cdot lat_{i_n} + b_{v_n} \\
&\Rightarrow lat_{i_n} = \frac{b_{v_n} - b_{l_1}}{m_{l_1} - m_{v_n}} \\
&\Rightarrow lon_{i_n} = m_{l_1} \cdot lat_{i_n} + b_{l_1}
\end{aligned}
$$

Afterward, we check if the intersection point i_n is between the two samples and the fence, which is the case for the intersection i_3, with:

$$\begin{aligned} & min(lat_{p_1}, lat_{p_2}) < lat_{i_n} < max(lat_{p_1}, lat_{p_2}) \\ \wedge\ & min(lon_{p_1}, lon_{p_2}) < lon_{i_n} < max(lon_{p_1}, lon_{p_2}) \\ \wedge\ & min(lat_{s_n}, lat_{s_{n+1}}) < lat_{i_n} < max(lat_{s_n}, lat_{s_{n+1}}) \\ \wedge\ & min(lon_{s_n}, lon_{s_{n+1}}) < lon_{i_n} < max(lon_{s_n}, lon_{s_{n+1}}) \end{aligned}$$

RTLola Specification

The specification in Figure 6.2 represents the previous example in RTLola. Because the geofence points p_1 and p_2 are fixed, we can precompute the function $l_1(lat) = m_{l_1} \cdot lat + b_{l_1}$ as well as the minimum and the maximum of the latitude and longitude of p_1 and p_2. In the specification, we represent these numbers with the italic variables.

At first we start with the declaration of the input streams `lat_in_degree` and `lon_in_degree` of type `Float32`. These streams represent the current latitude and longitude of the vehicle in degree. Because the following specification uses radians, the first two output streams transform the degree value in radians.

Next, we define the vehicle line between two samples. We compute the current gradient `m_v` and the y-intercept `b_v`, out of `lat`, `lat_pre`, `lon`, and `lon_pre` as in the previous example. However, two consecutive samples may have the same latitude value, which would result in a division by zero. For this, we define the output stream `is_Fnc`, comparing the latitude values to prevent the zero division. Such a line would correspond to a parallel line to the y-axis, which cannot be expressed mathematically but can result in a border crossing. Because the latitude and the longitude values have a floating-point representation in the data package, checking for equality is problematic, because two mathematically equal numbers may have different bit representations. So, instead of equality, we check if the difference between two numbers is smaller than a threshold ε. If the `is_fnc` stream is true, we assign the affected streams `m_v` and `b_v` to a default value 0. Afterward, we compute the current minimal and maximal latitude and longitude of the vehicle with the streams `min_lat_v`, `max_lat_v`, `min_lon_v`, and `max_lon_v`.

To check the crossing between a geofence line and the vehicle line, we compute their intersection. For this, we compute the latitude value by dividing the y-intercept difference with the gradient difference. This can result in a division by zero if the gradient is the same, i.e. if both lines are parallel to each other. In this case, there is no intersection, and so no geofence crossing, which is checked by the stream `intersect_p1p2`. With the latitude, we compute the longitude by inserting the value in the line function. Afterward, the trigger analyses if the existing intersect is in the bounds of the vehicle line and the border, which triggers an alarm with a crossing. In case that the vehicle line runs in

```
import math

// Latitude and Longitude in Degree
input lat_in_degree :Float32
input lon_in_degree :Float32

// Transform Degree in Radian
output lat := lat_in_degree * 3.14159265359 / 180.0
output lon := lon_in_degree * 3.14159265359 / 180.0

// Compute Vehicle Line
    // Samples for the Line Computation
output lat_pre := lat.offset(by: -1).defaults(to: lat)
output delta_lat := lat - lat_pre

output lon_pre := lon.offset(by: -1).defaults(to: lon)
output delta_lon := lon - lon_pre

    // Gradient and y-Intercept
output is_fnc := abs(delta_lat) > $\varepsilon$
output m_v := if isFnc then (delta_lon) / (delta_lat) else 0.0
output b_v := if isFnc then lon - (m_v * lat) else 0.0

    // Minimum and Maximum
output min_lat_v:= if lat < lat_pre then lat else lat_pre
output max_lat_v := if lat > lat_pre then lat else lat_pre

output min_lon_v := if lon < lon_pre then lon else lon_pre
output max_lon_v := if lon > lon_pre then lon else lon_pre

// Polygonline p1p2
output intersect_p1p2 := abs(m_v - $m_{l_1}$) > $\varepsilon$
output intersect_lat_p1p2
  := if is_fnc ∧ intersect_p1p2 then (b_v - $b_{l_1}$) / ($m_{l_1}$ - m_v) else lat
output intersect_lon_p1p2 := $m_{l_1}$ * intersect_lat_p1p2 + $b_{l_1}$
trigger intersect_p1p2
  ∧ ((intersect_lat_p1p2 > min_lat_v ∧ intersect_lat_p1p2 < max_lat_v)
  ∧ (intersect_lon_p1p2 > min_lon_v ∧ intersect_lon_p1p2 < max_lon_v))
  ∧ ((intersect_lat_p1p2 > $min_{lat_{p1},lat_{p2}}$ ∧ intersect_lat_p1p2 < $max_{lat_{p1},lat_{p2}}$)
  ∧ (intersect_lon_p1p2 > $min_{lon_{p1},lon_{p2}}$ ∧ intersect_lon_p0p1 < $max_{lon_{p1},lon_{p2}}$))
  "Border crossing between p1 and p2"
```

Figure 6.2.: RTLola specification to detect a line crossing in the geofence.

parallel to the y-axis, carried by the `is_fnc` stream, the latitude of the intersect is the current position of the vehicle, which cannot be computed from the function. This case is covered with the `if`-condition in `intersect_lat_p1p2`.

6.1.2. Sensor Validation

In the second experiment, we monitor the correctness of sensor values. In autonomous flights, the system uses the sensor values for navigation. For this, it is necessary that each sensor delivers reliable data. Incorrect data can be caused, for example, if the environment interfere the communication between the GPS Module and the satellites. To compute the current position, the module requests the position of GPS satellites to compute the trilateration out of their response time. If too less satellites are in range or if the response signal is blocked or even worse deflected, the module computes a false position, which the monitor has to detect. Another problem can be caused if the timing behavior of a module is unsatisfied, i.e. if the module delays it indicates a black out of the module or inaccurate data. We separate the experiment in two parts, with different concepts of how we monitor the incoming sensor values. At first, we monitor two packages from the Global Navigation Satellite System (GNSS), where we test the asynchronous behavior of RTLola. We define constraints, which validate the incoming events from the `GPSVelocity` package and events from the `GPSPosition` package in an event-based and time-based way. Additionally, we validate the timing behavior of the `GPSPosition` package. The second specification focuses on the validation of sensor values with respect to their timing behavior. In this part, we describe a property that can detect a significant change of a sensor value — called a peak — which is an indication for false sensor values.

Simple Sensor Validation

For this part of the case study, we use the specification in Figure 6.3. At first, we specify two constraints validating the incoming horizontal and vertical speed values from the `GPSVelocity` package against upper bounds. To represent these input speed values, we define the input streams `speed_h` and `speed_v`. Afterward, we specify the trigger checking the input values against their upper bounds and report the user if the system violates its speed limits. Then, we specify the constraints for the `GPSPosition` input values, separated into two parts. On the one hand, we validate the reliability of the sensor values, on the other hand the timing constraints. First, we monitor the enum `postion_type`. This enum defines the algorithm computing the GPS Position and has, in our setting, only two valid values. To check this constraint, we define a trigger reporting if the input stream `position_type` contains an invalid enum value. Next, we monitor the number of satellites used for the GPS-position computation. If the number of satellites decreases, the accuracy of the current position also decreases. In general, it is not alarming if this number decreases from time to time. However, with an increased number of violation over a small duration, the position becomes inaccurate. This bad situation should be detected by the monitor. For this, we define the output stream

```
1  import math
2
3  // GPS Velocity Package
4  input speed_h : Float16  // Horizontal Speed
5  input speed_v : Float16 // Vertical Speed
6
7  trigger abs(speed_h) > 1.5 "GPSVel Speed Horizontal"
8  trigger abs(speed_v) > 2.0 "GPSVel Speed Vertical"
9
10 // GPS Position Package:
11 input diff_age : Float32 // Time since Last Correction
12 input solution_age : Float16 // Time of the Computation
13 input position_type : UInt8 // Method of the Computation
14 input num_of_sats : UInt8 // Number of Satellites used for the Computation
15
16 trigger ¬(positionType = 34 ∨ positionType = 17) "False Position Type"
17
18 output cur_violation_with_sats_num : UInt8 := if numOfObs < 9 then 1 else 0
19 output violation_with_sats_num : Bool @1Hz
20   := cur_violation_with_sats_num.aggregate(over: 5s, using: sum) > 12
21 trigger ¬violation_with_sats_num.offset(by:-1).defaults(to:false) ∧
       violation_with_sats_num
22   "To less Satellites in Range"
23
24 trigger diff_age > 135.0 "Correction too old"
25 trigger solution_age > 0.15 "Computation too long"
26
27 trigger @1Hz num_of_sats.aggregate(over: 3s, using: count) < 10
28   "Frequency of GPS Module Violated"
```

Figure 6.3.: RTLola specification for a simple sensor validation, monitoring the GPS-module.

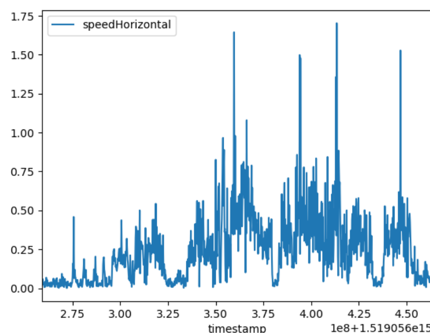


Figure 6.4.: Horizontal speed values from a test flight

`cur_violation_with_sats_num`, which compares the current number of observations against a threshold. Then, we aggregate the number of violations and trigger an alarm if necessary.

The last three triggers check the timing constraints of the sensor. One way to check such constraints are event-based streams like the first two triggers in line 24 and 25. These streams use the time-stamp produced by the sensor or system and compare them against upper bounds. The monitored UAV uses a differential GPS-Module, which considers for the position estimation the distance to an accurate DGPS Reference Station besides the satellite distances. The GPS-Module requests these stations frequently to improve the position accuracy. As a consequence, with increased time, the accuracy decreases. The time since the last correction is received in the `diff_age` stream, which the trigger compares against an upper bound. The second trigger compares the `sol_age` stream containing the period of how long the computation for the GPS module took. If this value is too large, the GPS is delayed. Event-based streams have the advantage of a very memory efficient check because the monitor only has to compare two values, which do not need to be stored. However, the monitor has to receive an incoming event to perform this check, which can become a problem. For example, the case that a sensor fails, and produces no further events, cannot be recognized by the monitor. Another possibility to check timing constraints is using periodic streams as the trigger in line 27. This stream counts the number of received events to compute the current frequency of the package. Because the monitor evaluates periodic streams with a frequency, the previous example is recognized. However, aggregating over a stream is not as memory efficient as comparing two values, resulting in one disadvantage of this approach. Another problem with this specification results from the architecture of the system. The event is produced by the sensor and is afterward sent by the system to the monitor. This architecture produces a variable delay depending on the system state of the processor sending the event, which results in false-positive alarms.

```
1  import math
2
3  input speed_h : Float16 // Horizontal Speed
4  input speed_v : Float16 // Vertical Speed
5
6  output avg_speed_h @1Hz
7    := speed_h.aggregate(over: 10s, using: avg).defaults(to:0.0)
8  output speed_h_diff
9    := abs(speed_h - avg_speed_h.hold().defaults(to:speed_h))
10 trigger speedH_diff > 0.4 "Peak in Horizontal Speed"
11
12 output avg_speed_v @1Hz
13   := speed_v.aggregate(over: 10s, using: avg).defaults(to:0.0)
14 output speed_v_diff
15   := abs(speed_v - avg_speed_v.hold().defaults(to:speed_v))
16 trigger abs_speed_v_diff > 1.0 "Peak in Vertical Speed"
17
18 output speed_all
19   := sqrt(speed_h * speed_h + speed_v * speed_v)
20 output avg_speed_all @1Hz
21   := speed_all.aggregate(over: 10s, using: avg).defaults(to:0.0)
22 output speed_all_diff
23   := abs(speed_all - avg_speed_all.hold().defaults(to:speed_all))
24 trigger speed_all_diff > 1.0 "Peak in Speed Vector"
```

Figure 6.5.: Specification for a peak detection, monitoring the horizontal and vertical speed.

Peak Detection

In the previous specification, we compared the velocity values against upper bounds. These checks however do not take the previous values into account, with is focused in this part of the sensor validation. As an example, consider the `speed_h` from Figure 6.4. We can see that at the beginning of the execution, the upper bound from the previous specification in Figure 6.3 is not violated. However, we also recognize that one data point differs a lot compared to the previous ones. We call such a significant difference a peak, which might have different causes, e.g. a GPS jump. Because the velocity is computed out of the GPS position, incorrect GPS coordinates from the GPS Module result in incorrect velocity values. To detect a peak, we need to consider a set of values beside the current one.

Such a peak detection in RTLola is represented in Figure 6.5, where the incoming velocity values from the `GPSVelocity` package are monitored. At first, the specification defines the input stream with the incoming velocity. Then, we specify a peak detection for the horizontal velocity, the vertical velocity, and the total velocity. The specification first computes a reference value with the output stream `avg_speed_h` to detect a peak of the horizontal speed. For this, we aggregate over the incoming events computing the average. Then, the next output stream `speed_h_diff` compares the current value with the reference. Because `speed_h` is event-based and `avg_speed_h` is periodic, we use the sample & hold lookup to access the current average value As a consequence, the stream `speed_h_diff` is evaluated for each incoming event. The following trigger then raises the alarm if this value is higher than a threshold. The same approach is used for the vertical speed. For the total speed, we first define the output stream `speed_all`, which computes the total velocity by calculating the length of the velocity vector. Then, we specify the streams to detect a peak for the total velocity. Alternatively to the periodic streams `avg_speed_h`, `avg_speed_v`, and `avg_speed_all`, we could also consider an event-based approach using offset values. However, in this case, the timing of the incoming events would not be included.

6.1.3. Cross Validation in RTLola

The last experiment defines the concept of a cross validation. In comparison to the experiment for the sensor validation, we do not compare the stream values against static threshold separately. Different modules have a connection to each other, e.g. if the acceleration increases this has an affection on velocity of the vehicle. If the behavior of the sensor is reflected in another sensor the confidence of both sensor value rises. In this theses, we compare the acceleration from the Internal Measurement Unit (IMU), with the velocity given by the GNSS. To compare these values, we integrate the acceleration after the time resulting in the velocity. Alternatively, we could also derivate the velocity, which give the acceleration.

Cross Validation in RTLola

Before we describe the RTLola specification for this experiment, we first debate the best stream lookups in RTLola for cross validations. Section 3.1.1 introduced synchronous lookups, sample & hold lookups and sliding window lookups. In the following examples, we address these three expressions and discuss the advantages and disadvantages. For this, we assume that the total velocity values from the IMU and GNSS are received with the input streams gps_vel and imu_vel.

➔Sec. 3.1, P. 8

First, we compare the input values with synchronous lookups resulting in the following specification:

```
input gps_vel : Float32
input imu_vel : Float32

output cross : Float32 := abs(gps_vel - imu_vel) < ε
```

With the synchronous lookups in cross, the activation condition of this event-based stream is bounded to the activation of gps_vel and imu_vel. As a consequence, if an incoming event never contains values from gps_vel and imu_vel simultaneously, the stream cross is never evaluated. In our setup, this is however the case. The IMU and the GNSS send their values in different packages, such that the activation condition of cross is never satisfied.

To solve this issue in RTLola, we prefer asynchronous lookups, starting with a sample & hold lookup.

```
input gps_vel : Float32
input imu_vel : Float32

output cross : Float32 := abs(gps_vel - imu_vel.hold().defaults(to: gps_vel)) < ε
```

Again we compare the current velocity form the GPS Module with the current velocity from the IMU. However, the imu_vel lookup is realized with a sample & hold lookup. This bounds the activation condition of the cross stream to the activation condition of the gps_vel stream, instead of both input streams. So cross is evaluated with every event updating gps_vel. This approach just compares the current values to each other. This logic needs hardly no resources, but only works in an optimal world, where the increase acceleration is directly reflected in the velocity. However, in practice the value of the models might be delayed or influenced with noise, but the general behavior of the sensor behave as expected. For this reason, we use for the cross validation sliding window lookups resulting in the specification:

```
input gps_vel : Float32
input imu_vel : Float32

output avg_gps_vel : Float32 @nHz
  := gps_vel.aggregate(over: d, using: avg).defaults(to: 0)
output avg_imu_vel : Float32 @nHz
  := imu_vel.aggregate(over: d, using: avg).defaults(to: 0)
output cross : Float32 := abs(avg_gps_vel - avg_imu_vel) < ε
```

First, the monitor computes with the streams avg_gps_vel and avg_imu_vel the average velocity of both streams. Then the cross stream compares these average values. This filters the small differences caused by the undesired delay and noise. Because the activation condition of all output streams is bounded to the same fixed frequency, we make sure that the cross stream is activated, comparing the velocity values. Instead of computing the average, we could also consider, the summation of all stream values. Depended on the value type of the stream and the duration of the window, summing all values has a high risk of an overflow. With the moving average approach, this risk is reduced to a minimum. Alternatively, we could increase the value type to a larger bit representation to prevent the overflow, but this increases the used memory consumption. Resources on the board limited are limited, such that we prefer the average computation.

Underlying Model

The specification in our experiment validates the GNSS against the IMU. For this check, the GNSS delivers the velocity of the vehicle in m/s. The IMU sends the acceleration of the vehicle in m/s^2. To get the velocity from the acceleration as demanded in the previous section, we need to integrate the acceleration. With the prototype the integration over a sliding window is realized with the trapezoid abstraction. This results in the RTLola specifciation:

```
input gps_vel : Float32
input imu_acc : Float32

output avg_gps_vel : Float32 @nHz
  := gps_vel.aggregate(over: d, using: avg).defaults(to: 0)
output imu_vel : Float32 @nHz
  := imu_acc.aggregate(over: d, using: ∫).defaults(to: 0)
output avg_imu_vel : Float32 @nHz
  := imu_vel.aggregate(over: d, using: avg).defaults(to: 0)
output cross : Float32 := abs(avg_gps_vel - avg_imu_vel) < ε
```

If we unroll the computation for the integral, we get the following formula for an input trace $t = ((a_1, t_1)\dots(a_n, t_n))$:

$$\sum_{1 \leq i < n-1} \frac{1}{2} \cdot (a_i + a_{i+1}) \cdot (t_{i+1} - t_i)$$

With this summation, we again have a high risk of an overflow, as described in the previous section. For this, we need another approach. To reduces this risk, we use the assumption that the input values are received with a fixed frequency, resulting in the input trace: $t = ((a_1, t_1)\dots(a_n, t_n)), \forall i.(t_{i+1} - t_i) = p$. If we apply this assumption to the trapezoid construction we get:

$$\sum_{1 \leq i < n} \frac{1}{2} \cdot (a_i + a_{i+1}) \cdot (t_{i+1} - t_i) = \sum_{1 \leq i < n} \frac{1}{2} \cdot (a_i + a_{i+1}) \cdot p$$

$$
\begin{aligned}
&= \frac{1}{2} \cdot p \cdot \Big(\sum_{1 \le i < n} (a_i + a_{i+1}) \Big) \\
&= \frac{1}{2} \cdot p \cdot \Big(\Big(\sum_{1 \le i < n} a_i \Big) + \Big(\sum_{1 \le i < n} a_{i+1} \Big) \Big) \\
&= \frac{1}{2} \cdot p \cdot \Big(a_1 + \Big(\sum_{2 \le i < n} a_i \Big) + \Big(\sum_{2 \le i < n} a_i \Big) + a_n \Big) \\
&= \frac{1}{2} \cdot p \cdot \Big(2 \cdot \Big(\sum_{2 \le i < n} a_i \Big) + a_1 + a_n \Big) \\
&= p \cdot \Big(\frac{1}{2} \cdot 2 \cdot \Big(\sum_{2 \le i < n} a_i \Big) + \frac{1}{2} \cdot (a_1 + a_n) \Big) \\
&= p \cdot \Big(\Big(\sum_{2 \le i < n} a_i \Big) + \frac{a_1}{2} + \frac{a_n}{2} \Big) \\
&\approx p \cdot \Big(\sum_{1 \le i \le n} a_i \Big) && \text{(for long traces } t\text{)}
\end{aligned}
$$

To get the average velocity, we divide the result with the number of incoming events n:

$$
\frac{p \cdot \Big(\sum_{1 \le i \le} a_i \Big)}{n} = p \cdot \frac{\Big(\sum_{1 \le i \le n} a_i \Big)}{n}
$$

This expression can be realized with the stream expression:

```
output avg_imu_vel : Float32 @nHz
  := p * imu_acc.aggregate(over: d, using: avg).defaults(to: 0)
```

Specification

Figure 6.6 represents the specification, which is used in the experiment. The specification receives from the IMU the acceleration in x, y, and z-direction. To get the total acceleration, the output stream acc computes the length of the acceleration vector. The `avg_IMU_vel` then integrates over the acceleration with the previously describes approach.

From the GPS module, the monitor uses the horizontal and vertical speed. Because the integration over the acceleration computes the relative velocity with respect to the start of the window instead of the absolute velocity, we cannot aggregate over the absolute speed values from the GPS module. For this, we aggregate over the speed difference instead of the absolute values. These transformations are realized with the output streams `speedH_diff` and `speed_v_diff`. As for the acceleration, we then compute the length of the vector with the `all_speed` stream and aggregate over this stream to get the average velocity. With the final trigger, we check if the difference between the IMU and the GPS Module is greater than a threshold, which raises an alarm if appropriate.

```
import math

// IMU
input acc_x : Float32 // Acceleration in x Direction
input acc_y : Float32 // Acceleration in y Direction
input acc_z : Float32 // Acceleration in z Direction

output acc := sqrt((acc_x * acc_x) + (acc_y * acc_y) + (acc_z * acc_z))
output avg_IMU_vel @1Hz := acc.aggregate(over: 10s, using: avg).defaults(to:0.0) * 0.01

// GPS Module
input speed_h : Float16 // Horizontal Speed
input speed_v : Float16 // Vertical Speed
output speed_h_diff : Float32 := cast(speed_h -
    speed_h.offset(by:-1).defaults(to:speed_h))
output speed_v_diff : Float32 := cast(speed_v -
    speed_v.offset(by:-1).defaults(to:speed_v))
output all_speed := sqrt(speed_h_diff * speed_h_diff + speed_v_diff * speed_v_diff)
output gpsVel_avg_vel @1Hz := all_speed.aggregate(over: 10s, using: avg).defaults(to:0.0)

// Comparison
trigger abs(gpsVel_avg_vel - avg_IMU_vel) > 0.5 "Cross Validation"
```

Figure 6.6.: Specification for a cross validation, comparing the acceleration with the velocity.

Specification/Module	FF	LUT	CA	MULT	Idle [mW]	Peak [W]	WNS [ns]
Geofence					149	1.871	9.011
Monitor	2,853	26,181	5,425	46			
HLC	504	275	38	0			
Queue	426	286	28	0			
LLC	1,721	25,429	5,359	46			
Sensor Validation					156	2.088	9.063
Monitor	4,800	34,356	8,480	128			
HLC	708	347	38	0			
Queue	627	600	52	0			
LLC	3,055	33,200	8,390	128			
Cross Validation					150	1.911	9.128
Monitor	3,441	23,261	5,703	100			
HLC	706	347	38	0			
Queue	627	475	44	0			
LLC	1,795	22,184	5,621	100			

Table 6.7.: Static Analysis

6.2. Evaluation

→Chap. 4, P. 53
→Chap. 5, P. 85
→App. A.5, P. 133

To validate the monitor, we compile three RTLola specifications to VHDL with the prototype from Chapter 4 and integrate the resulting monitor into the replay environment from Chapter 5. The first specification is a geofence from Section 6.1.1, consisting of 12 borders, visualized in Figure 6.10. For further details, Appendix A.5 presents the complete specification. The following specification is the union of the specifications from Section 6.1.2 containing the simple sensor validation and the peak detection. The third specification is the cross validation from Section 6.1.3.

6.2.1. Static Analysis

After synthesizing the VHDL code onto the ZC702 Base Board, we can statically analyze the resource consumption of the specifications, represented in Table 6.7. This table reports the number of flip-flops (FF), the number of lookup tables (LUT), the number of carry adders (CA), the number of multipliers (MULT), and the worst negative slack time (WNS).

Remark 6.2.1. *Flip-flops are used to store signal values for the next clock cycle. Lookup tables are a boolean operation table, which implements a boolean function. Carry adders and multipliers are gates to realize a fast binary addition and multiplication. The worst negative slack time is the time difference between the time when the next data is received and the time which is needed to process the longest path.*

In general, we see that in all three specifications, most of the resources are used in the LLC. This is expected because the LLC performs the stream evaluations, which is more complex than evaluating the activation conditions. An indicator of the complexity of the components is the number of lookup tables. The smallest difference is in the Cross Validation specification between the Queue and the LLC. There, the number of lookup tables is about 47 times higher than in the Queue. The most significant difference is in the Sensor Validation specification, where the number of lookup tables in the LLC is with a factor of 96 times higher than in the HLC. The different complexity of the components is also reflected in the number of carry adders and multipliers. In all specifications, the number of lookup tables differ from a factor of 47 up to a factor of 96. The different complexity between LLC, HLC, and Queue respectively is also reflected in the number of carry adders and multipliers.

Concerning the memory consumption of the specifications, we consider the number of flips flops. Table 6.7 shows that all specifications required more than 600B. As for the complexity, the memory consumption between the LLC and the HLC or LLC and Queue varies. Again, this is expected. The LLC stores all needed stream values, including the input and output streams, as well as the buckets in the sliding windows. If we compare this to the HLC, this component only contains the received event and the static array for the periodic streams, which are much fewer values than all stream or bucket entries. This is similar to the Queue, which stores beside the received event the enable bits of the output streams. For this, the number of flip-flops in the LLC is in every specification at least twice as many as the number of flip-flops in the HLC and Queue. If we compare the HLC with the Queue, we see that they have similar results.

To identify which streams need most of the resources, Table 6.8 reports the resource consumption of all entities in the geofence specification individually. First, we see that only the stream entities with a multiplication operation in the stream expression contain one or more multiplier components, which was expected. The same holds for the sliding windows aggregations. Concerning the lookup tables and the carry adders, we realize that most of the lookup tables and carry adders are used if the stream expression or sliding window contains a division operator. In the geofence specification, these are the streams computing the latitude position of the intersect and the `m_v` stream computing the current gradient of the vehicle line. This hypothesis is supported in the sensor validation and in the cross-validation specification, where the sliding windows computing the average need most of the resources. For example if we compare the sliding windows in the sensor validation specification, the `speed_all.aggregate(over: 10s, using: avg)` needs 10496 lookup tables, whereas `cur_violation_with_sats_num.aggregate(over: 5s, using: sum)` need 85 and `num_of_sats.aggregates(over: 3s, using: count)` uses 272. This high resource consumption for the division is logical. To compute the division,

Component	FF	LUT	CA	MULT
Monitor	2853	26181	5425	46
HLC	504	275	38	0
CheckNewInput	2	127	0	0
ExtInterface	66	0	0	0
EventDelay	128	8	0	0
TimeSelect	61	63	16	0
Scheduler	111	13	22	0
HLQInterface	133	61	0	0
Queue	426	286	28	0
LLC	1721	25429	5359	46
Evaluator	1717	25429	5359	46
LatInDegree	67	386	0	0
LonInDegree	67	193	0	0
Lat	68	841	104	4
Lon	68	903	104	4
LatPre	34	69	0	0
LonPre	34	68	0	0
DeltaLat	34	295	38	0
DeltaLon	34	102	22	0
IsFnc	6	8	8	0
M_V	46	3619	1000	0
b_V	34	366	116	4 0
MinLatV	34	195	4	0
MinLonV	34	181	4	0
MaxLatV	34	157	4	0
MaxLonV	34	161	4	0
IntersectP0P1	3	35	8	0
IntersectP1P2	3	33	8	0
IntersectP2P3	3	100	16	0
IntersectP3P4	3	66	8	0
IntersectP4P5	3	68	8	0
IntersectP5P6	3	65	8	0
IntersectP6P7	3	34	8	0
IntersectP7P8	3	63	8	0
IntersectP8P9	3	91	22	0
IntersectP9P10	3	33	8	0
IntersectP10P11	3	32	8	0
IntersectP11P12	3	64	8	0

Component	FF	LUT	CA	MULT
IntersectLatP0P1	34	2472	633	0
IntersectLatP1P2	34	2566	633	0
IntersectLatP2P3	34	2575	632	0
IntersectLatP3P4	34	199	10	0
IntersectLatP4P5	34	394	19	0
IntersectLatP5P6	34	2579	633	0
IntersectLatP6P7	34	2592	633	0
IntersectLatP7P8	34	321	19	0
IntersectLatP8P9	34	351	19	0
IntersectLatP9P10	34	336	19	0
IntersectLatP10P11	34	326	29	0
IntersectLatP11P12	34	420	19	0
IntersectLonP0P1	34	179	18	4
IntersectLonP1P2	34	130	8	2
IntersectLonP2P3	34	180	17	4
IntersectLonP3P4	34	133	8	2
IntersectLonP4P5	34	144	8	2
IntersectLonP5P6	34	178	18	4
IntersectLonP6P7	34	152	8	2
IntersectLonP7P8	34	155	18	4
IntersectLonP8P9	34	177	18	4
IntersectLonP9P10	34	156	8	2
IntersectLonP10P11	34	137	8	2
IntersectLonP11P12	34	124	8	2
TriggerP0P1	3	7	32	0
TriggerP1P2	3	4	32	0
TriggerP2P3	3	4	32	0
TriggerP3P4	3	5	32	0
TriggerP4P5	3	5	32	0
TriggerP5P6	3	6	32	0
TriggerP6P7	3	6	32	0
TriggerP7P8	3	4	32	0
TriggerP8P9	3	6	32	0
TriggerP9P10	3	4	32	0
TriggerP10P11	3	7	32	0
TriggerP11P12	3	5	32	0
Counter	68	71	8	0
(TimeCounter)	68	71	8	0

Table 6.8.: Resource consumption of the geofence specification.

the circuit needs several clock cycles. However, with a pipeline architecture, some computations can be parallelized to reduce the clock cycles but with the tradeoff of an increased resource consumption[1]. Further details about the resource consumption of the other specifications are presented in Appendix A.6.

→ App. A.6, P. 136

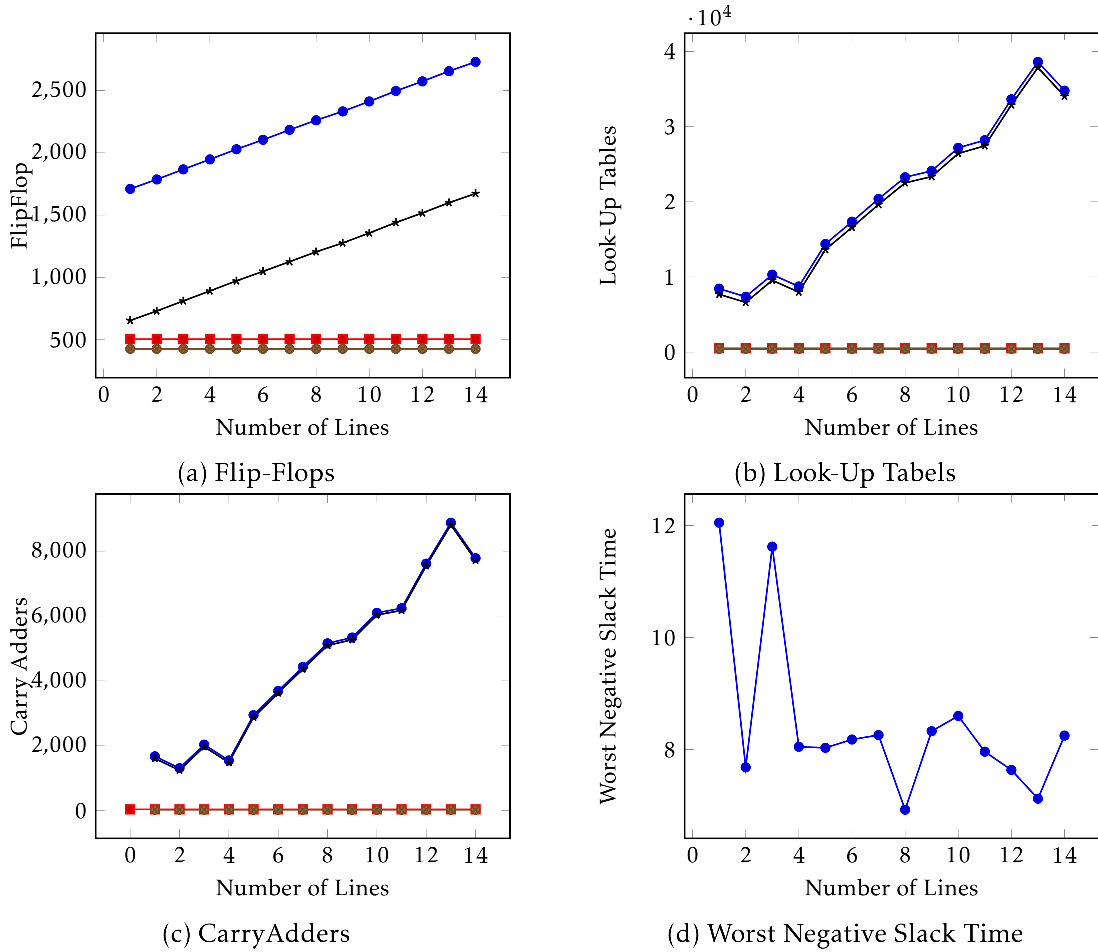
Remark 6.2.2. *In concern of the sliding windows, we have to involve the different value types and the duration of the window, such that the difference between the count and the sum aggregation can be explained. However, because the value type of the average sliding window is a 32-bit representation, whereas the counting aggregation uses a 64-bit representation, the example supports the hypothesis that division needs many resources.*

Concerning the worst negative slack time, which is also represented in Table 6.7, all specifications have a positive number such that the execution on the FPGA fulfills its timing constraints.

Table 6.8 also leads to another interesting fact, if we look at the `intersect_lat_p5p6` and `intersect_latp7p8` stream entities. Both streams contain the same stream expressions except different static constants. From a theoretical point of view, we would assume that the resource consumption of both streams would be equal. This assumption holds for the number of flip flops, where both streams contain 34 flip-flops. However, if we look at the number of lookup tables or carry bits, we see a huge difference. The stream entity `intersect_lat_p5p6` needs 2579 lookup tables and 633 carry adders, whereas `intersect_latp7p8` needs only 312 lookup tables and 19 carry adders. One explanation for this difference could be the multi-dimensional optimization of the `VHDL` synthesis. The synthesizer tries to parallelize as most computations as possible. From a theoretical point of view, this is possible for all border crossing computation. However, the space on the hardware-board is limited such that the synthesizer tries to reuse computation logic as long as the general timing behavior of the FPGA is still satisfied. This explains why the number of flip-flops are in both streams the same because the streams can contain different values, whereas the number of lookup tables differs such that previous computation logics are reused.

To check the behavior of specifications with different border numbers, we compile and synthesize a geofence with 14 borders, which was the maximum on the hardware board, and decreased the borders iteratively. Figure 6.9 summarizes the results for the number of flip-flops, lookup tables, and carry adders as well as the worst negative slack time. In these graphics, the blue line represents the consumption of the complete monitor, the red line the HLC, the brown line the Queue, and the black line the LLC. We see that for all specifications, the resource consumption of the HLC, and the Queue is equal. However, the consumption of the LLC and so forth the monitor varies. This was expected, because the logic for the HLC and the Queue is in all specifications the same. Figure 6.9a shows that the number of flip-flops increases linearly, which is unsurprising concerning the observations from the previous paragraph. Figure 6.9c and Figure 6.9b illustrate the number of lookup tables and carry adders. Due to the

[1] The synthesizer unrolls the division in our monitors, Radix-2 Solution in https://www.xilinx.com/support/documentation/ip_documentation/div_gen/v5_1/pg151-div-gen.pdf



(a) Flip-Flops

(b) Look-Up Tabels

(c) CarryAdders

(d) Worst Negative Slack Time

Figure 6.9.: Result of the static analysis for different number of lines in the geo-fence. The blue line visualizes the complete monitor, the red line the HLC, the brown line the Queue, and the black line the LLC.

previous observations, we assume a nonlinear increase of the lookup tables and carry adders, because the synthesis tool optimizes for a fast parallel execution but also for an execution with few resources by reusing logic computations. The tradeoff is seen if we compare the results of the specification with one border and with two borders. The number of lookup tables is in the specification with two borders smaller than with one, so the synthesized execution reuses some computation logic. For this, the worst negative slack time shown in Figure 6.9d, also decreases, expressing the less parallel execution. For specifications between five and seven borders, the number of carry adders and lookup tables increases linearly. For this reason, the worst negative slack time equals despite a small error margin.From the specification with eight borders, we cannot see such a pattern anymore. This can be explained by the different optimizations performed by the synthesizer. With the increased size of the architecture, it is harder

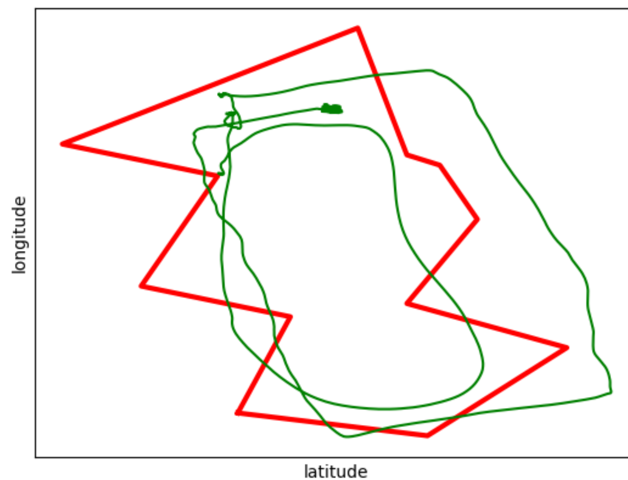


Figure 6.10.: Representation of the test flight and the geofence

to find interplays between the different entities optimizing the design. This explains why the worst negative slack time decreases between 10 and 13, even if the number of lookup tables increases.

Another static analysis is the power consumption of the monitor, shown in Table 6.7. All specifications have extremely low power consumption: In the idle state, they consume approximately 150mW and under heavy workload, a peak with approximately 2W. For comparison, we report the power behavior of two Raspberry PI model[2], where a software-based monitoring approach could run on. The Model 2B with s 32-bit architecture consumes at idle 1.1W and the Model 4B with a 64-bit architecture 2.7W. Under pressure, the power about twice as much with 2.1W and 6.4W. To justify the low power consumption of the prototype, we compare the values to a Mac Pro 2013 model[3]. The consumption in idle state is of factor 280 higher, where the consumption under pressure shows a factor 100 compared to our implementation

6.2.2. Validation

To evaluate the specification, we replayed two different flights with the setup from Chapter 5. The first benchmark is a test flight, represented with the green line in Figure 6.10. In this benchmark, the setup for the optical navigation was tested with different states of the vehicle,e.g. hovering. We monitored this flight with the geofence specification, visualized with the red line in Figure 6.10. The GPS position was sent from the GPS-Module with a frequency of 5Hz. During the flight, the vehicle crossed

→Chap. 5, P. 85

[2]Data collected from https://www.pidramble.com/wiki/benchmarks/power-consumption
[3]Data collected from https://support.apple.com/el-gr/HT201796

the fence twelve times on several lines, which the monitor recognized and reported. The second benchmark is an on-site ground test on the area of the DLR institute in Brunswick. In this test, the general setup of the landmark recognition was tested. For this, the vehicle is attached on a platform and moved manually. Because the test area is surrounded by buildings, the GPS-module had to deal with much interference. This was the motivation to verify the Sensor Validation and the Cross Validation with this run. For the Sensor Validation, the monitor received the GPS velocity data package, containing the speed data, and the GPS position data. Both packages were sent with a frequency of 5Hz from the GPS module. Our monitor reported 113 violations. However, most of them were connected, i.e. if the monitor recognized a peak on the horizontal speed, it often recognized a peak for the vertical and total speed as well. For the Cross Validation, the IMU sends the acceleration values with a frequency of 100Hz and the GPS module the speed values with a frequency of 5Hz. This monitor issued 36 violations.

Chapter 7

Conclusion & Future Work

This thesis presented a traceable compilation for RTLola specifications to VHDL. RTLola is a stream-based specification language with the great tradeoff-between formal guarantees and expressiveness. For this reason, it is an excellent candidate for formal runtime monitoring of cyber-physical systems. Safety-critical domains such as avionics, automotive, or medical markets share the restriction that these products are required to satisfy certification requirements. By showing the traceability property for our hardware-based approach, we made the first step to satisfy these requirements. In our prototype, we relate each component in the compiled architecture with the corresponding part in the specification and annotate them.

In the next step, we experimented with our prototype in a real-world domain, aviation. In cooperation with the German Aerospace Center in Brunswick, we integrated a hardware board into the replay environment of an unmanned aerial vehicle (UAV) and the UAV itself. The replayer played pre-recorded flights for the UAV with the same timing-behavior as the UAV, which provided a perfect development environment for the integration of the monitor. The integrated board contained a Field Programmable Gate Array (FPGA), on which we synthesized compiled RTLola specifications. As input data, the monitor received UDP packages from the UAV or replayer, analyzed these inputs and stored the output of the monitor into a file. In the resulting case study, we tested the integration and the compilation in a real-world domain with realistic frequencies. We described three RTLola specifications which covered different parts of the vehicle: The sensor validation checked the correctness of the sensor data. The cross-validation described different approaches to validate different sensors against each other. The geofence specification monitored a high-level command structure of the vehicle by analyzing the position of the vehicle and checking if the UAV stays in the restricted area, described by a polygon. We synthesized all specifications on the hardware board and performed different static analyzes. The power consumption of each monitor was around 150mW in idle and around 2W under pressure, which is extremely low. We also reported the resource consumption of the specifications, which was on the limit of the hardware board. The geofence specification presented that the synthesis tool reuses

some computations to decrease the number of resources, but results in less parallel execution. Additionally, we integrated the compiled monitors in the replay environment. We replayed two pre-recorded flights, violating different constraints, which the monitor recognized. Unfortunately, we could not validate the monitor during a flight because of timing schedule problems and weather conditions. However, a flight test with the hardware-based monitor is scheduled in March 2020, to identify the overall system impact.

Currently, the hardware board stores the stream values into a file and does not provide any feedback during the execution. For further applications, the stream values and triggers need to be provided to the pilot or the system. In this case, the monitor supports the pilot by providing compressed data and notifications in case of a violation. Alternatively, the output of the monitor can be provided to the system directly, which can start counter measurements automatically, e.g. starting a safe landing. However, in this case, the output of the monitor needs to be considered when designing and developing the UAV.

The specifications in this thesis were on the limit of resources of the hardware-board, but only partially monitored the UAV. This issue can be solved with distributed monitoring. In this case, we split the logic of the specification into different monitors. For this approach, an architecture needs to describe the inputs and outputs of the separate monitors as well as their connections. An example of an architecture can be a separation between the sensor validations as well as the cross-validation: The validation of the different sensors is covered by separate monitors, which send compressed data to the cross-validation. This approach solves the problem of the board limitations but introduces new ones, e.g. the encoding of the architecture in the RTLola specification, or representing the time for the communication between the monitors.

Besides, the language RTLola is not yet complete. For example, one extension of Lola [34] introduces parameterization for network monitoring. In this case, the number of instances of a parametrized stream is bounded by the value types of the parameters. This results in a conflict when using our hardware-based approach: The number of entities needs to be known at compile-time, to realize the logic in the hardware. Therefore, we need to define an instance for each parameter value. However, because of the board limitation, we cannot create all instances. One approach to solve this issue is to restrict the number of instances explicitly instead of using the value types of the parameter. However, in this case, we need to define the semantics if the number of instances is violated with an input trace.

Appendix A

Appendix

A.1. Inference Rules Type System

Input Streams

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

Periodic Output Streams

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

Event-base Output Streams

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

Trigger

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

Synchronous Lookups

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

Offset Lookups

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

Sample & Hold Lookup

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

Default Expression

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

Function Expression

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

Sliding Window Lookup

$$\frac{\delta \in \natural \quad \gamma : T_a^* \rightarrow T_r \quad \tau \sqsubseteq lift(T_r) \quad \widetilde{\sigma}', \widetilde{\tau}' \models s_i^- \quad \widetilde{\tau}' \sqsubseteq lift(T_a)}{\widetilde{\pi}, \widetilde{\tau} \models Window(s_i^-, \delta, \gamma)}$$

A.2. Templates

A.2.1. TimeSelect

The code fragment realizes the computation of the system time:

```
1  -- Internal Signal Declarations
2  signal sys_time : unsigned(63 downto 0);
3
4  begin
5
6  process(clk, rst) begin
7    if rst = '1' then
8      -- Reset Phase
9      sys_time <= to_unsigned(0, sys_time'length);
10   elsif rising_edge(clk) then
11     -- Logic Phase: Compute System Time
12     -- Relation Clock Frequency and Period per Cycle:
13     -- {{clk_freq_in_hz}} <=> {{period_in_sec}}
14     sys_time <= sys_time + {{time_per_cycle}};
15   end if;
```

```
16 end process;
```

A.2.2. CheckNewInput

This entity realizes the detection when the values on the input signals of the HLC entity are a new incoming event for the monitor. For this, the entity waits on a rising edge on the new_input signal, set by the monitored system.

```
1  -- Internal Signal Declarations
2  signal prev_new_input_in : std_logic;
3
4  begin
5
6  process(clk, rst) begin
7    if rst = '1' then
8      -- Reset Phase
9      new_input_out <= '0';
10     prev_new_input_in <= '0';
11   elsif rising_edge(clk) then
12     -- Logic Phase: Check If There Is a New Event
13     if new_input_in = '1' and prev_new_input_in = '0' then
14       -- Current Event Is New
15       new_input_out <= '1';
16     else
17       -- No New Event
18       new_input_out <= '0';
19     end if;
20     prev_new_input_in <= new_input_in;
21   end if;
22 end process;
```

A.2.3. ExtInterface

This entity realizes the conversion from the bit-vectors to the corresponding numeric types.

```
1  --* Input Streams and their Types in the Specification:
2  --* {{print_input_streams}}
3
4  -- Internal Signal Declarations
5  signal time_converted : unsigned(63 downto 0);
6  {{converted_signals}}
7
8  begin
```

```
9
10 process(clk, rst) begin
11     if rst = '1' then
12         -- Reset Phase
13         time_converted <= to_unsigned(0, time_converted'length);
14         {{signal_default_assignment}}
15     elsif rising_edge(clk) then
16         -- Logic Phase: Convert Input in Numeric Types
17         time_converted <= unsigned(time_in);
18         {{converts}}
19     end if;
20 end process;
```

A.2.4. HLC

This process creates a clock that is fourth times slower, such that the HLQInterface can forward the information of the Scheduler and the EventDelay in one clock cycle of the HLC.

```
1  process(clk, rst) begin
2    if (rst = '1') then
3      -- Reset Phase
4      slow_hlc_clk <= '0';
5      hlc_clk_count <= 0;
6    elsif rising_edge(clk) then
7      -- Logic Phase: Raise Slow Clock Signal Every Fourth Cycle
8      hlc_clk_count <= (hlc_clk_count + 1) mod 4;
9      if hlc_clk_count = 3 then
10        slow_hlc_clk <= '1';
11     else
12        slow_hlc_clk <= '0';
13     end if;
14   end if;
15 end process;
```

A.2.5. LLQInterface

This process realizes the state machine from Figure 3.10.

```
1  process(clk, rst) begin
2    if rst='1' then
3      -- Reset Phase
4      eval <= '0';
5      current_state <= 0;
6      pop_data <= '0';
```

```
7      elsif rising_edge(clk) then
8        -- Logic Phase
9        if (current_state = 0 and data_available = '1') then
10         -- idle
11         pop_data <= '1';
12         eval <= '0';
13         current_state <= 1;
14       elsif current_state = 1 then
15         -- pop
16         eval <= '1';
17         pop_data <= '0';
18         current_state <= 2;
19       elsif current_state = 2 and evaluator_done = '1' then
20         -- eval
21         if data_available = '1' then
22           pop_data <= '1';
23           eval <= '0';
24           current_state <= 1;
25         else
26           eval <= '0';
27           current_state <= 0;
28         end if;
29       end if;
30     end if;
31   end process;
```

A.3. Realization of Stream Expressions

To realize stream expression in the prototype, we assign each subexpression to a new VHDL variable. Such variables are declared after the process keyword and have a scope limited to this single process. In comparison to signal assignments, which perform their assignments in parallel, variable assignments have a sequential execution. This method increases the depth of the resulting circuit and slows down the clock signal. However, with this sequential execution, we can evaluate a stream expression in one clock cycle and do not have to separate the evaluation. The compiler uses the abstract syntax tree (AST) of a stream expression and assigns each subexpression to a temporary variable. This AST is part of the intermediate representation (IR) of the specification, returned by the StreamLab frontend. Our prototype compiles the corresponding VHDL realization from this tree by iterating through the AST in post-order and assigning each subexpression to a temporary variable. With these temporal assignments, the prototype can compile the VHDL code for each expression type separately without using the previous context. The following algorithm gives an idea of the recursive approach for the expression generation.

First, the algorithm checks the amount of children of the current expression. Afterward, it builds recursively the VHDL realization for each child. Then, the compiler

Algorithm 1: generate_vhdl_code_for_expression

Data: The parameter *expr* contains the current expression.
Result: The function returns the code fragment realizing the stream expression.

```
1  if expr is leave expression then
2  |  return generate_leave_expression(leave type of expr);
3  if expr is unary expression then
4  |  subexpr = generate_vhdl_code_for_expression(subexpression of expr);
5  |  return subexpr + generate_unary_expression(unary type of expr);
6  if expr is binary expression then
7  |  lhs = generate_vhdl_code_for_expression(lhs of expr);
8  |  rhs = generate_vhdl_code_for_expression(rhs of expr);
9  |  return lhs + rhs + generate_binary_expression(binary type of expr);
10 if expr is ternary expression then
11 |  para_1 = generate_vhdl_code_for_expression(first parameter of expr);
12 |  para_2 = generate_vhdl_code_for_expression(second parameter of expr);
13 |  para_3 = generate_vhdl_code_for_expression(third parameter of expr);
14 |  return para_1 + para_2 + para_3 + generate_ternary_expression(ternary type
   |    of expr);
15 return Error;
```

realizes the VHDL code of the current expression type and returns the code fragments of the subexpressions and the current expression type. Sometimes, the prototype needs more than one temporary variable for the realization of one expression type. To see the realization for each expression, we describe the expression in vanilla RTLola and show their realizations:

LoadConstant(constant)

A constant c in StreamLab is a function f, which maps to the constant value c. Therefore, the result of the *AST* function in vanilla RTLola is: $Func(f)$, which need to be transformed in VHDL code. Based on the type T of the constant the prototype builds the following realization:

Stream Type	VHDL Realization	Explanation
Bool	$\texttt{temp}_{const}$:= c	c' is '1' if the constant value c is true and '0' otherwise
UInt8, UInt16, UInt32 or UInt64	$\texttt{temp}_{const}$:= to_unsigned(c,n-1)	n is the size of the UInt type
Int8, Int16, UInt32 or UInt64	$\texttt{temp}_{const}$:= to_signed(c,n-1)	n is the size of the Int type
Float16, Float32 or Float64	$\texttt{temp}_{const}$:= to_sfixed(c,l,u)	l is the lower bound and u is the upper bound of the Float type

StreamAccess(target, kind)

A stream access to s_i^- in the IR is either a synchronous lookup or a hold access. Therefore the output of the *AST* function would be *Offset*(s_i^-, 0) or *Hold*(s_i^-). Due to the parsing that checks if a default expression follows the hold access, there is no difference for the IR. In both cases, the expression generator assigns the corresponding input value to the temporary variable.

```
temp_acc := s_i^-;
```

OffsetLookup(target, offset)

The prototype realizes an offset lookup *Offset*(s_i^-, n) by assigning the input signal from the output stream entity to the temporary variable. Like the realization of a hold access, the compiler knows at this time that a default expression follows an offset lookup. Therefore, there is no check if the value is valid at this position, and the corresponding VHDL code is:

```
temp_off := s_i^-_negn;
```

WindowLookup(window)

The realization of a window lookup *Window*(s_l^-, δ, γ) is the same as a synchronous access to another stream. The compiler builds a new entity for each sliding window computing the sliding window. The value is received by the output stream entity and assigned to the temporary variable, resulting in:

```
temp_wl := s_i^-_δ_γ_sw_d;
```

`Default(sub_expr,default)`

For the realization, the expression generation function receives the last offset $Offset(s_i^-, n)$, hold $Hold(s_i^-)$, or sliding window access $Window(s_l^-, \delta, \gamma)$ from the recursive call. Additionally, a variable *last_access* is needed to access the correct `valid` input signal. With this value, the function decides if to take the default value or the expression value. For this decision the compiler generates:

```
temp_def := sel(temp_sub_expr, temp_default, valid_last_access);
```

where $\texttt{temp}_{sub_expr}$ is assigned to the subexpression value and $\texttt{temp}_{default}$ is assigned to the default value. Additionally, the function `sel` returns the first value, if the third argument is `'1'` and the second otherwise.

`Ite(condition, consequence, alternative)`

This expression is syntactic sugar for a function expression $Func(f, cond, cons, alt)$. The definition of f is to return the second parameter *cons* if the first one containing the condition *cond* is true and the third one with the alternative *alt* otherwise. Therefore, the first parameter is of type `Bool`, and the second and third one have the same type t. In StreamLab, the three parameters are expressions that are realized by a recursive call of the expression generator. As a result, the subexpressions are assigned to the temporary variables $\texttt{temp}_{cond}$, $\texttt{temp}_{cons}$ and $\texttt{temp}_{alt}$. To realize the `Ite` expression itself the function generates the following code fragment:

```
if temp_cond = '1' then
  temp_ite := temp_cons;
else
  temp_ite := temp_alt;
endif;
```

`ArithLog(operator, parameters)`

In vanilla RTLola, the arithmetic operators are represented as functions. Based on the operator, these functions have a different number of parameters. The realization uses the corresponding `VHDL` operator and the variables containing the subexpressions for the assignment. However, dependent on the operator or the stream type of the expression, the `VHDL` realization needs to resize the variable afterwards to their corresponding bit length in the specification. The following table shows for each operator the realization, where u and l define the same range as in Section 4.2.

→ Sec. 4.2, P. 56

Operation	Result Type	VHDL Realization	Notes
$-expr$		`temp`$_{res}$ `:= - temp`$_{expr}$`;`	
$\neg expr$		`temp`$_{res}$ `:= neg temp`$_{expr}$`;`	
$lhs \circ rhs$	Integer	`temp`$_{res}$ `:= temp`$_{lhs}$ `∘ temp`$_{rhs}$`;`	$\circ \in \{+,-,/\}$
$lhs \circ rhs$	Fix-point Number	`temp`$_{op}$ `:= temp`$_{lhs}$ `∘ temp`$_{rhs}$`;` `temp`$_{res}$ `:= temp`$_{op}$`(`u `downto` l`);`	$\circ \in \{+,-,/\}$
$lhs \,\%\, rhs$		`temp`$_{res}$ `:= temp`$_{lhs}$ `rem temp`$_{rhs}$`;`	
$lhs * rhs$		`temp`$_{*}$ `:= temp`$_{lhs}$ `* temp`$_{rhs}$`;` `temp`$_{res}$ `:= temp`$_{*}$`(`u `downto` l`);`	
$lhs \wedge rhs$		`temp`$_{res}$ `:= temp`$_{lhs}$ `and temp`$_{rhs}$`;`	
$lhs \vee rhs$		`temp`$_{res}$ `:= temp`$_{lhs}$ `or temp`$_{rhs}$`;`	
$lhs \circ rhs$		`temp`$_{res}$ `:= to_std_logic(temp`$_{lhs}$ `∘ temp`$_{rhs}$`);`	$\circ \in \{=,<,>,$ $>=,<=\}$
$lhs \neq rhs$		`temp`$_{res}$ `:= to_std_logic(temp`$_{lhs}$ `/= temp`$_{rhs}$`);`	

`Function(name, parameters, returntype)`

Our prototype supports at this point in time two functions apart from the arithmetic operators, the *abs* returning the absolut value of a number and the square root function *sqrt*. To compute the absolut value in VHDL, we use the predefined abs function from the ieee library, resulting in the following assignments:

```
-- integer type
temp_res := abs(temp_expr);
-- Real Number with Fix-point Representation
temp_abs := abs(temp_expr);
temp_res := temp_expr(u downto l);
```

with u is the upper bound and l is the lower bound from the fix-point representation from Section 4.2. As for some of the arithmetic operators, we have to resize the bit vector after applying the operation.

→Sec. 4.2, P. 56

In VHDL there is no predefined square root function in the ieee library for the numeric types signed, unsigned and sfixed. Currently, the frontend in StreamLab enforces that the argument type of the function is a float type. Because of the fix-point representation of real numbers, we can use integer arithmetic and only need to implement a square root function for integers. We realized the square root computation for unsigned integer with

the constant-time function proposed by Li and Chu [35]. To implement the function for real numbers, we interpret the bit vector as a real number and divide the result afterwards based on the representation. The following proof shows the correctness of this approach, where x is a real number with the bit representation using m bits for the fractional part.

$$\begin{aligned}\sqrt{x} &= \sqrt{\frac{x * 2^{m+1}}{2^{m+1}}} \\ &= \frac{\sqrt{x * 2^{m+1}}}{\sqrt{2^{m+1}}} \\ &= \frac{\sqrt{unsigned(x) << 1}}{2^{\frac{m+1}{2}}} \\ &= \left(\sqrt{unsigned(x) << 1}\right) >> \frac{m+1}{2}\end{aligned}$$

Since the *sqrt* function is only defined for positive numbers, the first bit of x is always zero. Therefore, by multiplying the number with 2 after interpreting the bit vector as an unsigned value, we ensure that there was no overflow, because the first bit is zero. This shift is needed, because Section 4.2 assigns each stream type for real number to an odd number m, and otherwise $\frac{m+1}{2}$ would not be an integer value.

The implementation of this approach is outsourced into a new `VHDL` package — called `my_math_pkg` — which is included in each output stream entity . The expression generator only calls the implemented function with the temporary variable, containing the result of the sub expression *expr* and assigns it to a new one:

```
temp_res := my_sqrt_func(temp_expr);
```

Convert(from_type, to_type, expr)

The conversion of the expression *expr* of type t_{from} to another type t_{to} is separated into two categories. The first kind of conversion is a resizing of the same type, e.g. the type `Int8` is converted to `Int32`. Because no type cast is needed for this category, the realization is a mapping of the corresponding bits. The compiler determines the upper bound u and lower bound l, from the type with the smaller bit range and assigns only these bits to the new temporary variable:

```
temp_res(u downto l) := temp_expr(u downto l);
```

The second kind of conversion involves a new interpretation of the bit vector. At first, the signed interpretation is cancelled by uninterpreting the bit vector. Afterwards, the bit vector is resized if needed and interpreted as an unsigned integer value:

```
temp_bitvec(u downto l) := std_logic_vector(temp_expr(u downto l));
temp_res := unsigned(temp_bitvec);
```

For a conversion from an unsigned integer value to a signed one, the compiler uses the same approach and replaces the `unsigned` keyword with `signed`. If one of the types is a real number, the expression generator uses the functions `to_sfixed` and `to_signed` from the `fixed_pkg` in the `ieee` library. These function take as inputs the expression, that needs to be converted and the new bit range. For a conversion from a real number to an integer with an unsigned interpretation and vice versa, the realization uses an additional step with a conversion to an integer value with a signed interpretation:

```
-- From Int to Float
temp_res := to_sfixed(temp_expr, u, l);
-- From UInt to Float
temp_res := to_sfixed(signed(temp_expr), u, l);
-- From Float to Int
temp_res := to_signed(temp_expr, u);
-- From Float to UInt
temp_res := unsigned(to_signed(temp_expr, u));
```

where u and l are the upper and lower bounds of the t_{to} type.

A.3.1. Sliding Window Realizations

Summation

To add up all values over a time window, we use the aggregation function $sum : A^* \to A$

- $map_{sum} : A \to A$ with $map_{sum}(x) = x$
- $fin_{sum} : A \to A$ with $fin_{sum}(x) = x$
- $\circledast_{sum} : A^2 \to A$ with $x_1 \circledast_{sum} x_2 = x_1 + x_2$
- $\varepsilon_{sum} = 0$

Apart from the lifting function map_{sum}, this homomorphism has the same definition as the counting aggregation. The realization of this function is similar to the counting realization. At first, we change the name of the buckets register to `sum_buckets`. Afterwards, the compilation replaces the `{{map_and_update_last_sw_bucket}}` place holder with:

```
sum_buckets(0) <= sum_buckets(0) + d_in;
```

For real numbers, the compiler resizes the bit vector after the addition like in Section **??**. →Sec. 0, P. ??

Averaging

The computation of the average value over some time sums and counts all values and divides the sum by the count during the finalization. Therefore, the list homomorphism uses a tuple for the intermediate representation to store the sum and the count value. The mathematical description for the averaging aggregation $avg : A^* \to A$ is:

- $map_{avg} : A \to (A, \mathbb{N})$ with $map_{avg}(x) = (x, 1)$

- $fin_{avg} : (A, \mathbb{N}) \rightarrow A$ with $fin_{avg}(x, c) = \frac{x}{c}$
- $\circledast_{avg} : (A, \mathbb{N})^2 \rightarrow A$ with $(x_1, c_1) \circledast_{avg} (x_2, c_2) = (x_1 + x_2, c_1 + c_2)$
- $\varepsilon_{avg} = (0, 0)$

To realize the tuple used as the intermediate representation, the prototype uses two internal registers for the bucket approach, which initialize each bucket entry to zero. The `count_buckets` signal counts the number of received stream values per bucket and the `sum_buckets` signal stores the summed value for each bucket. The realization of the averaging aggregation *avg* is then the combination of the realization for the counting and summation function for the evict and the update case. Then, the request-case replaces `{{finalize_sw}}` with the separate summation of the arrays and performs the division:

```
d <= (sum_bucket(0) + ... + sum_bucket({{num_buckets}})
 / (count_bucket(0) + ... + count_bucket({{num_buckets}}));
```

Like in the summation, the compiler adds a resize operation before writing the value to the data signal. Since the division by zero is not defined, which is possible, if no value was received inside the timed window, the entity notifies the stream expression to take the default value. Therefore, the return value for the `d_valid_out` is assigned to `'0'`, which is achieved by replacing the `{{valid_upd}}` place holder with `'0'`. This assignment defines each bucket entry to invalid at first. If an event is received, meaning a rising edge of the `upd` signal, the valid entry for this bucket is assigned to one. Therefore, the compiler adds the following assignment to the `map_and_update_last_sw_bucket` replacement:

```
data_valid_buckets(0) <= '1';
```

In the request case, the compiler builds the disjunction of all `valid_buckets` entries for `{{finialize_valid}}`.

Integration

The integration over a specific time is performed in the prototype with the trapezoid abstraction. This approach reconstructs the function from received samples by connecting them with a straight line and computes the volume of this reconstructed graph. The sub-aggregations and, therefore, the different buckets in the homomorphism represent a section of the graph. The intermediate value of the function is an optional tuple storing the left-most value and its timestamp, the right-most value and its time-stamp and the current volume. The complete homomorphism $\int : (A, T)^* \rightarrow Optional(A)$ is described by:

- $map_{\int} : (A, T) \rightarrow Optional(A, T, A, T, A)$ with $map_{\int}((x, t)) = (x, t, x, t, 0)$
- $fin_{\int} : Optional(A, T, A, T, A) \rightarrow Optional(A)$ with
  $fin_{\int}(\bot) = \bot$ and
  $fin_{\int}((x^L, t^L, x^R, t^R, v)) = v$

- $\circledast_\int : (Optional(A,T,A,T,A))^2 \rightarrow Optional(A,T,A,T,A)$ with
  $\bot \circledast_\int \bot = \bot$,
  $(x_1^L, t_1^L, x_1^R, t_1^R, v_1) \circledast_\int \bot = (x_1^L, t_1^L, x_1^R, t_1^R, v_1)$,
  $\bot \circledast_\int (x_2^L, t_2^L, x_2^R, t_2^R, v_2) = (x_2^L, t_2^L, x_2^R, t_2^R, v_2)$ and
  $(x_1^L, t_1^L, x_1^R, t_1^R, v_1) \circledast_\int (x_2^L, t_2^L, x_2^R, t_2^R, v_2)$
  $= (x_1^L, t_1^L, x_2^R, t_2^R, \frac{1}{2} \cdot (x_1^R + x_2^L) \cdot (t_2^L - t_1^R) + v_1 + v_2)$
- $\varepsilon_\int = \bot$

This function requires besides the stream value also the system time. However, this value is already available for the evict phase, and the update phase can reuse the `time_in` input signal. For the tuple realization, the implementation uses a single internal register for each component in the tuple representation:

- The signal `lhs_value_buckets` stores the first value of a bucket.
- The signal `rhs_value_buckets` stores the last value of a bucket.
- The signal `lhs_time_buckets` stores the time stamp of the first element in a bucket.
- The signal `rhs_time_buckets` stores the timestamp of the last element added to a bucket.
- The signal `volume_buckets` stores the current volume of a bucket.

Like averaging, integration is not defined if a window does not receive any value. Therefore, the buckets in the `valid_bucket` signal array are assigned to `'0'` when creating a new one and during the reset phase. For the mathematic function, this zero assignment corresponds to the $\bot$ value. The other bucket signals are assigned to their type-specific zero values to prevent an access to an uninitialized value. During the update case, the realization has to differentiate the execution with a $\bot$ value and none. During the execution, the $\bot$ value is only possible in the first iteration for a bucket, the update last bucket case starts with a case disjunction if the current bucket value is valid. If this is not the case, which happens in the first iteration, the implementation assigns the bucket to the current input and the volume value to `'0'`. Otherwise, the input is lifted to the intermediate representation and a binary reduction is performed. This logic results in the following code fragment for `{{map_and_update_last_sw_bucket}}`, where the variables `half_time_diff` and `product` are variables with a sequential execution:

```
if (data_valid_buckets(0) = '0') then
  lhs_value_buckets(0) <= d_in;
  lhs_time_buckets(0) <= time_in;
  rhs_value_buckets(0) <= d_in;
  rhs_time_buckets(0) <= time_in;
  volume_buckets(0) <= (others => '0');
else
  half_time_diff := std_logic_vector((time_in - rhs_time_bucket(0)) / 2 );
```

```
  product := (rhs_value_buckets(0) - d_in) * cast(time_diff) ;
  rhs_value_buckets(0) <= d_in;
  rhs_time_buckets(0) <= time_in;
  volume_buckets(0) <= volume_buckets(0) + resize(product);
end if;
```

cast is a VHDL code fragment to cast the time and the value to the same numeric type. *resize* is a code fragment that casts the product value to the right bit-vector size (Compare Section 4.2). The request case combines the different buckets with the binary reduction operator and finalizes the result by returning the volume value. The implementation uses an iterative to realize this logic. Starting with the last bucket, the implementation iterates over all buckets, and stores the pre-aggregated results in the pre_valid, last_lhs_value, last_rhs_time and cur_volume variables. Because the implementation iterates from the bucket with the latest timestamp to the earliest one, the right-hand side of the pre-aggregations can be ignored and do not need to be stored. This approach is realized by:

→Sec. 4.2, P. 56

```
-- Iterate Over All Buckets, Beginning with the Last Time Stamp
pre_valid := data_valid_buckets(0);
last_lhs_value := lhs_value_buckets(0);
last_lhs_time := lhs_time_buckets(0);
cur_volume := volume_buckets(0);
for i in 1 to {{num_buckets}} loop
  if pre_valid = '0' then
    -- Reduction with Bot for the Pre-aggregated Values
    pre_valid := data_valid_buckets(i);
    last_lhs_value := lhs_value_buckets(i);
    last_lhs_time := lhs_time_buckets(i);
    cur_volume := volume_buckets(i);
  elsif data_valid_buckets(i) = '1' then
    -- Reduction with Valid Values on Both Sides
    half_time_diff := std_logic_vector((last_lhs_time - rhs_time_buckets(i)) / 2
        );
    product := (last_lhs_value + rhs_value_buckets(i)) * cast(time_diff);
    cur_volume := cur_volume + volume_buckets(i) + * resize(product);
    last_lhs_value := lhs_value_buckets(i);
    last_lhs_time := lhs_time_buckets(i);
  end if;
  data <= cur_volume;
```

The {{finalize_valid}} place holder is assigned to the pre_valid variable, which is set during the iteration over the buckets.

A.4. Roadmap to Integrate the monitor in Vivado

This is a roadmap for a program, that creates a binary file for an runtime monitor for an RTLola specification, that can be executed onto an Xilinx board. This roadmap is based on the tutorials [36, 37]. We perform the following steps:

Step 1: Compile VHDL file out of an RTLola specification
Step 2: Create a new Vivado Project
Step 3: Create a new Block design
Step 4: Create Monitor Component
Step 5: Generate Hardware Bitstream
Step 6: Generate First Stage Boot Loader and Add lwip Library
Step 7: Generate the Execute Project
Step 8: Generate Boot Image

Step 1: Compile VHDL file out of an RTLola specification

- open a terminal and execute the **fpga_streamlab.exe** program in the following way: `fpga_streamlab.exe --vivado_files --<online|offline> <pathToSpecification> <pathToTargetDirectory> <pathToTemplateDirectory>`

Step 2: Create a new Vivado Project

- Run Vivado and create a new project
  - Click on **Create New Project**
  - Click on **Next**
  - Enter a project namen, location and click the **Next** botton
  - Select **RTL Project** and click **Next**
  - Click on **Next**
  - Click on **Boards** and select the **Zynq ZC702 Evaluation Board** file
  - Click on **Next**
  - Click on **Finish**
- Add Compiled Monitor:
  - Click on **Tools** → **Create and Package New IP**
  - Click on **Next**
  - Select *Create a new AXI4 peripheral* and click on **Next**
  - Enter a monitor name and click on **Next**
  - Enter a number of Register, whereas the output of `fpga_streamlab` gives you the minimum number and click on **Next**
  - Select *Edit IP* and click on **Next**
  - A new window should open

Step 3: Create a new Block design

- Creating a New Block Design
  - Open the window, that is **not** named `edit_<monitorname>`
  - Click on **Create Block Design**
  - Enter a name and click on **OK**
  - Click the **Add IP** button and search for the **ZYNQ7 Processing System**
  - Click on **Run Block Automation** and click on **OK**
  - Click the **Add IP** button and search for the *<monitorname>*
  - Click on **Run Block Automation** and click on **OK**
- Validate Design, Generate HDL Wrapper and Generate Bitstream
  - Click on **Validate Design**
  - Right-click on *design_name.bd* in the *Design Sources* tab and click on **Create HDL Wrapper**

Step 4: Create Monitor Component

- Add the compiles files
  - Open the window, that is named *edit_<monitorname>*
  - Click on **Add Sources**
  - Click on **Next**
  - Click several times on **Add Files** and add all compiled files except the files in the *vivado_files* folder
  - Click on **Finish**
- Add the *fixed_pkg_2008* package to the ieee library
  - Click on **Tcl Console**
  - Type the following commands:
    * `add_files -norecurse <path to pkg>/fixed_pkg_2008.vhd`
    * `set_property library ieee [get_files <path to pkg>/fixed_pkg_2008.vhd]`
    * `read_vhdl -vhdl2008 <path to ip_repo>edit.xpr`
    * `launch_runs synth_1 -jobs 4`
    * `wait_on_run synth_1`
- Integrate the monitor
  - Open the *0_S00_AXI_inst* file in the *Source* → *Design Sources* tab

  - Go to the line with the signal declaration for the registers and before the `begin` keyword
  - Open the *vivado_integration.vhdl* file in the *vivado_files* folder
  - Copy all line up to the `end component` keyword and paste the line at the previous described position
  - Move the curser at the beginning bracket after the process keyword after the following comment "Implement memory mapped register select and read logic generation"
  - Copy the next line from the *vivado_integration.vhdl* file at this position and add a comma
  - Replace the following case disjunction with the case disjunction from the *vivado_integration.vhdl* file
  - Go to the comment "Add user logic here" and copy the rest of the *vivado_integration.vhdl* file at this position
  - Store the file

- Update the package
  - Click on *Package IP* file
  - Click on **File Groups** and on **Merge changes**
  - Click on **Review and Package** and on **Re-Package IP**
  - Click on **Yes**

Step 5: Generate Hardware Bitstream

- Update Block Design
  - Click on **Report IP Status**
  - Click on **Upgrade Selected** in the *IP Status* Tab
  - Click on **Generate**
- Generate Bitstream
  - Click on **Generate Bitstream** at the Flow Navigator tab. Wait for the process and click **OK**
- Export Hardware File and Launch SDK
  - Go to *File* → *Export* → *Export Hardware*
  - Select the **Include bitstream** box and click on **OK**
  - Go to *File* → *Launch SDK* and click on **OK**

Step 6: Generate First Stage Boot Loader and Add lwip Library

- Generate the First Stage Boot Loader Project
  - Go to *File* → *New* → *Application Project*
  - Enter *fsbl* as project name and click on **Next**
  - Select the **Zynq FSBL** template and click on **Finish**
- Add the *lwip141* library to the project
  - Open the *system.mss* file in the *fsbl_bsp* folder
  - Click on the **Modify this BSP's Settings** button
  - Select the *lwip141* box in the *Overview* → *Supported Libraries* tab and click on **OK**

Step 7: Generate the Execute Project

- Generate the Echo Project
  - Go to *File* → *New* → *Application Project*
  - Enter a project name
  - Choose *Use existing* in the *Board Support Package* tab, select *fsbl_bsp* and click on **NEXT**
  - Select the **IwIP Echo Server** template and click on **Finish**
- Modify the Echo Template and Board Support Package
  - Delete the *echo.c* and *main.c* file
  - Add all **.h* and **.c* files from the *vivado_file* folder to the *src* folder
  - Select the *system.mss* file in the *fsbl_bsp* folder and click on **Modify this BSP's Settings**
  - Click on *Overview* → *standalone* → *lwip141* → *temac_adapter_options*, change the value *phy_link_speed* from *Autodetect* to *100 Mbps*, and click on **OK**
  - Correct the base address in the *macros.h* file

Step 8: Generate Boot Image

- Create Boot Image and Store It on the SD Card
  - Right-click on the *echo* folder and click on *Create Boot Image*
  - Click on **Create Image**
  - Copy the *BOOT.bin* file in the `.../projectname.sdk/execute/bootimage` folder to the SD card
- Test the program

- Open Tera for the serial communication
- Plug in the SD card in and change the switch setting (SW16) to `00110`
- Start the sender program

A.5. Geofence

```
import math
input lat_in_degree :Float32
input lon_in_degree :Float32

output lat := lat_in_degre * 3.14159265359 / 180.0
output lon := lon_degree * 3.14159265359 / 180.0
output lat_pre := lat.offset(by: -1).defaults(to: lat)
output delta_lat := lat - lat_pre
output min_lat := if lat < lat_pre then lat else lat_pre
output max_lat := if lat > lat_pre then lat else lat_pre
output lon_pre := lon.offset(by: -1).defaults(to: lon)
output delta_lon := lon - lon_pre
output min_lon := if lon < lon_pre then lon else lon_pre
output max_lon := if lon > lon_pre then lon else lon_pre
output isFnc := abs(delta_lat) > 0.00000001
output m := if isFnc then (delta_lon) / (delta_lat) else 0.0
output b := if isFnc then lon-(m*lat) else 0.0

// Polygonline p0p1: (0.1180559967662996, 0.0399734162336672) to (0.11908193655673749,
    0.04525405540696442)
output intersect_p0p1 := abs(m-5.147123859035988) > 0.00000001
output lat_p0p1 := if isFnc and intersect_p0p1 then (b - -0.5676754214244288) /
    (5.147123859035988 - m) else lat
output lon_p0p1 := 5.147123859035988 * lat_p0p1 + -0.5676754214244288
output check_p0p1 := (intersect_p0p1) and ((lat_p0p1 > min_lat and lat_p0p1 < max_lat)
    and (lon_p0p1 > min_lon & lon_p0p1 < max_lon)) and ((lat_p0p1 > 0.1180559967662996
    and lat_p0p1 < 0.11908193655673749) and (lon_p0p1 > 0.0399734162336672 & lon_p0p1 <
    0.04525405540696442))

// Polygonline p1p2: (0.11908193655673749, 0.04525405540696442) to (0.11617135825882872,
    0.046935706350154524)
output intersect_p1p2 := abs(m--0.5777721026774507) > 0.00000001
output lat_p1p2 := if isFnc and intersect_p1p2 then (b - 0.11405627628225343) /
    (-0.5777721026774507 - m) else lat
output lon_p1p2 := -0.5777721026774507 * lat_p1p2 + 0.11405627628225343
output check_p1p2 := (intersect_p1p2) and ((lat_p1p2 > min_lat and lat_p1p2 < max_lat)
    and (lon_p1p2 > min_lon & lon_p1p2 < max_lon)) and ((lat_p1p2 > 0.11617135825882872
    and lat_p1p2 < 0.11908193655673749) and (lon_p1p2 > 0.04525405540696442 & lon_p1p2 <
    0.046935706350154524))

// Polygonline p2p3: (0.11617135825882872, 0.046935706350154524) to (0.11766289020098507,
    0.05297933593688546)
output intersect_p2p3 := abs(m-4.051961219143263) > 0.00000001
```

```
output lat_p2p3 := if isFnc and intersect_p2p3 then (b - -0.42378613208981786) /
    (4.051961219143263 - m) else lat
output lon_p2p3 := 4.051961219143263 * lat_p2p3 + -0.42378613208981786
output check_p2p3 := (intersect_p2p3) and ((lat_p2p3 > min_lat and lat_p2p3 < max_lat)
    and (lon_p2p3 > min_lon & lon_p2p3 < max_lon)) and ((lat_p2p3 > 0.11617135825882872
    and lat_p2p3 < 0.11766289020098507) and (lon_p2p3 > 0.046935706350154524 & lon_p2p3
    < 0.05297933593688546))

// Polygonline p3p4: (0.11766289020098507, 0.05297933593688546) to (0.11463282400467073,
    0.054732465100919954)
output intersect_p3p4 := abs(m--0.5785778430078313) > 0.00000001
output lat_p3p4 := if isFnc and intersect_p3p4 then (b - 0.12105647715143869) /
    (-0.5785778430078313 - m) else lat
output lon_p3p4 := -0.5785778430078313 * lat_p3p4 + 0.12105647715143869
output check_p3p4 := (intersect_p3p4) and ((lat_p3p4 > min_lat and lat_p3p4 < max_lat)
    and (lon_p3p4 > min_lon & lon_p3p4 < max_lon)) and ((lat_p3p4 > 0.11463282400467073
    and lat_p3p4 < 0.11766289020098507) and (lon_p3p4 > 0.05297933593688546 & lon_p3p4 <
    0.054732465100919954))

// Polygonline p4p5: (0.11463282400467073, 0.054732465100919954) to (0.12039516145301407,
    0.06110106989155227)
output intersect_p4p5 := abs(m-1.1052120511379069) > 0.00000001
output lat_p4p5 := if isFnc and intersect_p4p5 then (b - -0.07196111344501288) /
    (1.1052120511379069 - m) else lat
output lon_p4p5 := 1.1052120511379069 * lat_p4p5 + -0.07196111344501288
output check_p4p5 := (intersect_p4p5) and ((lat_p4p5 > min_lat and lat_p4p5 < max_lat)
    and (lon_p4p5 > min_lon & lon_p4p5 < max_lon)) and ((lat_p4p5 > 0.11463282400467073
    and lat_p4p5 < 0.12039516145301407) and (lon_p4p5 > 0.054732465100919954 & lon_p4p5
    < 0.06110106989155227))

// Polygonline p5p6: (0.12039516145301407, 0.06110106989155227) to (0.12135284424675223,
    0.05413856066294947)
output intersect_p5p6 := abs(m--7.270162181180861) > 0.00000001
output lat_p5p6 := if isFnc and intersect_p5p6 then (b - 0.936393419484419) /
    (-7.270162181180861 - m) else lat
output lon_p5p6 := -7.270162181180861 * lat_p5p6 + 0.936393419484419
output check_p5p6 := (intersect_p5p6) and ((lat_p5p6 > min_lat and lat_p5p6 < max_lat)
    and (lon_p5p6 > min_lon & lon_p5p6 < max_lon)) and ((lat_p5p6 > 0.12039516145301407
    and lat_p5p6 < 0.12135284424675223) and (lon_p5p6 > 0.05413856066294947 & lon_p5p6 <
    0.06110106989155227))

// Polygonline p6p7: (0.12135284424675223, 0.05413856066294947) to (0.12199513900152693,
    0.05357314974496478)
output intersect_p6p7 := abs(m--0.8802982023152586) > 0.00000001
output lat_p6p7 := if isFnc and intersect_p6p7 then (b - 0.16096525129920902) /
    (-0.8802982023152586 - m) else lat
output lon_p6p7 := -0.8802982023152586 * lat_p6p7 + 0.16096525129920902
output check_p6p7 := (intersect_p6p7) and ((lat_p6p7 > min_lat and lat_p6p7 < max_lat)
    and (lon_p6p7 > min_lon & lon_p6p7 < max_lon)) and ((lat_p6p7 > 0.12135284424675223
    and lat_p6p7 < 0.12199513900152693) and (lon_p6p7 > 0.05357314974496478 & lon_p6p7 <
    0.05413856066294947))
```

```
// Polygonline p7p8: (0.12199513900152693, 0.05357314974496478) to (0.12272805380095549,
    0.05060231541569706)
output intersect_p7p8 := abs(m--4.053451140001608) > 0.00000001
output lat_p7p8 := if isFnc and intersect_p7p8 then (b - 0.5480744850053588) /
    (-4.053451140001608 - m) else lat
output lon_p7p8 := -4.053451140001608 * lat_p7p8 + 0.5480744850053588
output check_p7p8 := (intersect_p7p8) and ((lat_p7p8 > min_lat and lat_p7p8 < max_lat)
    and (lon_p7p8 > min_lon & lon_p7p8 < max_lon)) and ((lat_p7p8 > 0.12199513900152693
    and lat_p7p8 < 0.12272805380095549) and (lon_p7p8 > 0.05060231541569706 & lon_p7p8 <
    0.05357314974496478))

// Polygonline p8p9: (0.12272805380095549, 0.05060231541569706) to (0.12135284424675223,
    0.04598052457716757)
output intersect_p8p9 := abs(m-3.3607902333162474) > 0.00000001
output lat_p8p9 := if isFnc and intersect_p8p9 then (b - -0.3618609291524651) /
    (3.3607902333162474 - m) else lat
output lon_p8p9 := 3.3607902333162474 * lat_p8p9 + -0.3618609291524651
output check_p8p9 := (intersect_p8p9) and ((lat_p8p9 > min_lat and lat_p8p9 < max_lat)
    and (lon_p8p9 > min_lon & lon_p8p9 < max_lon)) and ((lat_p8p9 > 0.12135284424675223
    and lat_p8p9 < 0.12272805380095549) and (lon_p8p9 > 0.04598052457716757 & lon_p8p9 <
    0.05060231541569706))

// Polygonline p9p10: (0.12135284424675223, 0.04598052457716757) to (0.12447245465942572,
    0.043556374755305015)
output intersect_p9p10 := abs(m--0.777068127486174) > 0.00000001
output lat_p9p10 := if isFnc and intersect_p9p10 then (b - 0.14027995202111265) /
    (-0.777068127486174 - m) else lat
output lon_p9p10 := -0.777068127486174 * lat_p9p10 + 0.14027995202111265
output check_p9p10 := (intersect_p9p10) and ((lat_p9p10 > min_lat and lat_p9p10 <
    max_lat) and (lon_p9p10 > min_lon & lon_p9p10 < max_lon)) and ((lat_p9p10 >
    0.12135284424675223 and lat_p9p10 < 0.12447245465942572) and (lon_p9p10 >
    0.043556374755305015 & lon_p9p10 < 0.04598052457716757))

// Polygonline p10p11: (0.12447245465942572, 0.043556374755305015) to
    (0.12175508788375872, 0.03872383447446285)
output intersect_p10p11 := abs(m-1.7783908760921594) > 0.00000001
output lat_p10p11 := if isFnc and intersect_p10p11 then (b - -0.17780430293581267) /
    (1.7783908760921594 - m) else lat
output lon_p10p11 := 1.7783908760921594 * lat_p10p11 + -0.17780430293581267
output check_p10p11 := (intersect_p10p11) and ((lat_p10p11 > min_lat and lat_p10p11 <
    max_lat) and (lon_p10p11 > min_lon & lon_p10p11 < max_lon)) and ((lat_p10p11 >
    0.12175508788375872 and lat_p10p11 < 0.12447245465942572) and (lon_p10p11 >
    0.03872383447446285 & lon_p10p11 < 0.043556374755305015))

// Polygonline p11p12: (0.12175508788375872, 0.03872383447446285) to (0.1180559967662996,
    0.0399734162336672)
output intersect_p11p12 := abs(m--0.3378077802156663) > 0.00000001
output lat_p11p12 := if isFnc and intersect_p11p12 then (b - 0.07985365044243875) /
    (-0.3378077802156663 - m) else lat
output lon_p11p12 := -0.3378077802156663 * lat_p11p12 + 0.07985365044243875
output check_p11p12 := (intersect_p11p12) and ((lat_p11p12 > min_lat and lat_p11p12 <
    max_lat) and (lon_p11p12 > min_lon & lon_p11p12 < max_lon)) and ((lat_p11p12 >
```

```
    0.1180559967662996 and lat_p11p12 < 0.12175508788375872) and (lon_p11p12 >
    0.03872383447446285 & lon_p11p12 < 0.0399734162336672))

// Activates termination
//output any_violated := ! any_violated.offset(by:-1).defaults(to:false) and
    (height_check or check_p0p1 or check_p1p2 or check_p2p3 or check_p3p4 or check_p4p5
    or check_p5p6 or check_p6p7 or check_p7p8 or check_p8p9 orcheck_p9p10 or
    check_p10p11 or check_p11p12)

output counter : Int32:= counter.offset(by:-1).defaults(to:0) + 1
output time_counter : Int32 @1Hz := time_counter.offset(by:-1).defaults(to:0) + 1
```

A.6. Static Analyzes

A.6.1. Cross Validation

Component	FF	LUT	MUX	CA	MULT
Monitor	3441	23261	99	5703	100
HLC	706	347	0	38	0
CheckNewInput	2	194	0	0	0
ExtInterface	133	0	0	0	0
EventDelay	195	13	0	0	0
TimeSelect	61	63	0	16	0
Scheduler	111	68	0	22	0
HLQInterface	200	60	0	0	0
Queue	627	475	0	44	0
LLC	1795	22184	0	5621	100
Evaluator	1791	22184	0	5621	100
AllImuAccAvgSlidingWindow	731	10183	0	2620	0
AllImuAcc	18	871	0	156	12
speedAllAvgSlidingWindow	731	9917	0	2609	40
speedAll	18	806	0	146	8
speedAllAvg	34	131	0	8	0
speedHDiff	18	4	0	4	0
speedH	35	36	0	0	0
speedVDiff	18	4	0	4	0
speedV	35	33	0	0	0
accIMU0	33	1	0	0	0
accIMU1	33	1	0	0	0
accIMU2	33	1	0	0	0
avgSpeedIMU	29	182	0	58	0
TriggerCrossValidation	3	6	0	16	0

A.6.2. Sensor Validation

Component	FF	LUT	MUX	CA	MULT
Monitor	4800	34356	0	8480	128
HLC	708	347	0	38	0
CheckNewInput	3	195	0	0	0
EventDelay	196	1	0	0	
ExtInterface	134	0	0	0	
TimeSelect	61	63	0	16	0
Scheduler	111	68	0	22	0
HLQInterface	100	17	0	0	0
Queue	627	600	0	52	0
LLC	3055	33200	0	8390	128
Evaluator	3051	33200	0	8390	128
avgSpeedAll	35	69	0	16	0
avgSpeedHorizontal	35	36	0	0	0
avgSpeedVertical	35	36	0	0	0
CheckGPSNumOfObs	3	6	0	0	0
CheckGPSNumOfObsSumSliding Window	106	85	0	28	0
DiffAge	33	32	0	0	0
NumOfObsCountSlidingWindow	314	272	0	54	0
NumOfObs	9	7	0	0	0
posType	9	4	0	0	0
solAge	15	14	0	0	0
avgSpeedAllAvgSlidingWindow	731	9873	0	2598	40
speedAll	18	843	0	146	8
avgSpeedHAvgSlidingWindow	731	10496	0	2696	40
speedH	33	335	0	38	0
avgSpeedVAvgSlidingWindow	731	10499	0	2696	40
speedV	33	336	0	38	0
speedAllDiff	34	101	0	4	0
speedHDiff	34	36	0	12	0
speedVDiff	34	37	0	12	0
timeCheck	6	8	0	0	0
TriggerFreq	3	3	0	8	0
TriggerDiffAge	3	6	0	4	0
TriggerPosType	3	3	0	0	0
TriggerSolAge	3	7	0	2	0
TriggerspeedHBound	3	3	0	8	0
TriggerspeedHPeak	3	5	0	4	0
TriggerspeedVBound	3	4	0	8	0
TriggerspeedVPeak	3	4	0	4	0
TriggerNumofObs	3	5	0	0	0
TriggerSpeedAllBound	3	17	0	10	0
TriggerSpeedAllPeak	3	4	0	4	0

Bibliography

[1] Yliès Falcone, Klaus Havelund, and Giles Reger. 2013. A tutorial on runtime verification. In *Engineering Dependable Software Systems*. NATO Science for Peace and Security Series, D: Information and Communication Security. Volume 34. Manfred Broy, Doron A. Peled, and Georg Kalus, editors. IOS Press, 141–175. ISBN: 978-1-61499-206-6. DOI: 10.3233/978-1-61499-207-3-141. https://doi.org/10.3233/978-1-61499-207-3-141.

[2] Klaus Havelund and Allen Goldberg. 2005. 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* (Lecture Notes in Computer Science). Bertrand Meyer and Jim Woodcock, editors. Volume 4171. Springer, 374–383. ISBN: 978-3-540-69147-1. DOI: 10.1007/978-3-540-69149-5_40. https://doi.org/10.1007/978-3-540-69149-5_40.

[3] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78, 5, 293–303. DOI: 10.1016/j.jlap.2008.08.004. https://doi.org/10.1016/j.jlap.2008.08.004.

[4] Oleg Sokolsky, Klaus Havelund, and Insup Lee. 2012. Introduction to the special section on runtime verification. *STTT*, 14, 3, 243–247. DOI: 10.1007/s10009-011-0218-6. https://doi.org/10.1007/s10009-011-0218-6.

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

[6] Edmund M. Clarke and E. Allen Emerson. 2008. Design and synthesis of synchronization skeletons using branching time temporal logic. In *25 Years of Model Checking - History, Achievements, Perspectives* (Lecture Notes in Computer Science). Orna Grumberg and Helmut Veith, editors. Volume 5000. Springer, 196–215. ISBN: 978-3-540-69849-4. DOI: 10.1007/978-3-540-69850-0_12. https://doi.org/10.1007/978-3-540-69850-0_12.

[7] Ron Koymans. 1990. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2, 4, (October 1990), 255–299. issn: 0922-6443. doi: 10.1007/BF01995674. http://dx.doi.org/10.1007/BF01995674.

[8] Oded Maler and Dejan Nickovic. 2004. Monitoring temporal properties of continuous signals. In *In: Proceedings of FORMATS-FTRTFT. Volume 3253 of LNCS*. Springer, 152–166.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79, 9, 1305–1320. issn: 0018-9219. doi: 10.1109/5.97300.

[10] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. 2005. Lola: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, 166–174. doi: 10.1109/TIME.2005.26.

[11] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. 2010. Copilot: A hard real-time runtime monitor. In *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, 345–359. doi: 10.1007/978-3-642-16612-9_26. https://doi.org/10.1007/978-3-642-16612-9_26.

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

[13] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah. 2019. Streamlab: stream-based monitoring of cyber-physical systems. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, 421–431. doi: 10.1007/978-3-030-25540-4_24. https://doi.org/10.1007/978-3-030-25540-4_24.

[14] Jan Baumeister, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. 2019. FPGA stream-monitoring of real-time properties. In *EMSOFT 2019*.

[15] Moe Shahdad. 1986. An overview of VHDL language and technology. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference. Las Vegas, NV, USA, June, 1986*. Don Thomas, editor. IEEE Computer Society Press, 320–326. doi: 10.1145/318013.318063. https://doi.org/10.1145/318013.318063.

[16] Hong Lu and Alessandro Forin. 2007. The Design and Implementation of P2V, An Architecture for Zero-Overhead Online Verification of Software Programs. Technical report MSR-TR-2007-99, 12. https://www.microsoft.com/en-us/research/publication/the-design-and-implementation-of-p2v-an-architecture-for-zero-overhead-online-verification-of-software-programs/.

[17] Ping Hang Cheung and Alessandro Forin. 2007. A c-language binding for psl. In *Embedded Software and Systems*. Yann-Hang Lee, Heung-Nam Kim, Jong Kim, Yongwan Park, Laurence T. Yang, and Sung Won Kim, editors. Springer Berlin Heidelberg, Berlin, Heidelberg, 584–591.

[18] Bernd Finkbeiner and Lars Kuhtz. 2009. Monitor circuits for LTL with bounded and unbounded future. In *Proceedings of the 9th International Workshop on Runtime Verification (RV 2009)* (LNCS). Saddek Bensalem and Doron A. Peled, editors. Volume 5779. Springer, 60–75.

[19] Patrick Moosbrugger, Kristin Y. Rozier, and Johann Schumann. 2017. R2u2: monitoring and diagnosis of security threats for unmanned aerial systems. *Form. Methods Syst. Des.*, 51, 1, (August 2017), 31–61. issn: 0925-9856. doi: 10.1007/s10703-017-0275-x. https://doi.org/10.1007/s10703-017-0275-x.

[20] Kristin Yvonne Rozier and Johann Schumann. 2017. R2U2: tool overview. In *RV-CuBES* (Kalpa Publications in Computing). Volume 3. EasyChair, 138–156.

[21] N. Halbwachs. 2005. A synchronous language at work: the story of lustre. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE '05.* 3–11. doi: 10.1109/MEMCOD.2005.1487884.

[22] Lukas Convent, Sebastian Hungerecker, Torben Scheffel, Malte Schmitz, Daniel Thoma, and Alexander Weiss. 2018. Hardware-based runtime verification with embedded tracing units and stream processing. In *RV* (Lecture Notes in Computer Science). Volume 11237. Springer, 43–63.

[23] Felipe Gorostiaga and César Sánchez. 2018. Striver: stream runtime verification for real-time event-streams. In *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, 282–298. doi: 10.1007/978-3-030-03769-7_16. https://doi.org/10.1007/978-3-030-03769-7_16.

[24] Florian-Michael Adolf, Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Christoph Torens. 2017. Stream runtime monitoring on UAS. In *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*, 33–49. doi: 10.1007/978-3-319-67531-2_3. https://doi.org/10.1007/978-3-319-67531-2_3.

[25] Maximilian Schwenger. 2019. *Let's not Trust Experience Blindly: Formal Monitoring of Humans and other CPS*. Master Thesis. Saarland University. https://www.react.uni-saarland.de/publications/Schwenger19.pdf.

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

[27] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34, 1, 39–44. doi: 10.1145/1058150.1058158. https://doi.org/10.1145/1058150.1058158.

[28] 2020. *FPGA*. https://www.mikrocontroller.net/articles/FPGA.

[29] 2020. *What is an FPGA?* https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html.

[30] 2020. *Basics of FPGA Architecture and Applications*. https://www.elprocus.com/fpga-architecture-and-applications/.

[31] Dagan White and DTodd R. White. 2011. Practical use of fpgas and ip in do-254 compliant systems. https://www.xilinx.com/support/documentation/white_papers/wp403_DO254_IP_Use.pdf.

[32] *Visual Navigation for Autonomous, Precise and Safe Landing on Celestial Bodies using Unscented Kalman Filtering*, (). IEEE Aerospace Conference.

[33] Franz Andert, Nikolaus Alexander Ammann, Stefan Krause, Sven Lorenz, Dmitry Bratanov, and Luis Mejias Alvarez. 2017. Optical-aided aircraft navigation using decoupled visual slam with range sensor augmentation. *Journal of Intelligent & Robotic Systems*, 88, 2-4, 547–565.

[34] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. 2016. A stream-based specification language for network monitoring. In *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, 152–168. doi: 10.1007/978-3-319-46982-9_10. https://doi.org/10.1007/978-3-319-46982-9_10.

[35] Yamin Li and Wanming Chu. 1996. A new non-restoring square root algorithm and its VLSI implementation. In *1996 International Conference on Computer Design (ICCD '96), VLSI in Computers and Processors, October 7-9, 1996, Austin, TX, USA, Proceedings*, 538–544. doi: 10.1109/ICCD.1996.563604. https://doi.org/10.1109/ICCD.1996.563604.

[36] 2020. *Getting Started with Zynq Servers*. https://reference.digilentinc.com/learn/programmable-logic/tutorials/zybo-getting-started-with-zynq-server/start.

[37] 2020. *Styx : Boot from SD card and QSPI flash*. https://numato.com/kb/styx-boot-sd-card-qspi-flash/.