From RTLola to SRTLola: A Block-Based Syntax for Stream-Based Specification Languages

Saarland University

Department of Computer Science

Master's Thesis

submitted by

Julia Laichner

Saarbrücken, December 2024

Supervisor:	Prof. Bernd Finkbeiner, Ph.D.
Advisor:	Jan Baumeister, M.Sc.
	Florian Kohn, B.Sc.
Reviewer:	Prof. Bernd Finkbeiner, Ph.D.
	Asst. Prof. Mark Santolucito, Ph.D.
Submission:	December 18, 2024

Abstract

Cyber-physical systems (CPS) appear in various aspects of modern life, finding applications in medicine, industries, and domestic environments. Ensuring the safety of these systems is significant, as they induce risk to both humans and the environment.

Given their safety-critical nature, CPS must fulfill specific correctness properties, which are typically expressed with specification languages. However, writing correct specifications is challenging, especially for newcomers. Visual programming languages like Scratch provide a user-friendly graphical interface where constructing programs is simplified by a block-based syntax using drag-and-drop components. Such tools reduce the cognitive load and enhance overall usability. Currently, these approaches exist primarily for imperative programming languages.

This thesis presents a visualization environment that uses a block-based syntax for stream-based specification languages, using RTLola as an example. This tool illustrates the RTLola language with a block-based syntax, allowing users to create a visual representation of the specification. Moreover, users can translate the block-based specification into the text-based one. With this approach, we strive to lower the entry barrier for future users to learn the RTLola language and create a more user-friendly developing environment for CPS.

Acknowledgements

First of all, I am grateful to Prof. Finkbeiner for giving me the opportunity to work on this exciting topic in my thesis. Further, I thank Mark Santolucito for reviewing this thesis. Moreover, I am grateful for all the help from my advisors, Jan Baumeister and Florian Kohn, who offered me continuous support and guidance throughout this thesis. I want to thank all the participants of the user studies for their time and effort. Lastly, a special thanks to Angelina and Niklas for proofreading my thesis and for their support and encouragement.

Saarbrücken, 18 December, 2024

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

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

Saarbrücken, 18 December, 2024

Contents

Chapter 1

Introduction

Learning a new programming language is a challenging task, especially for beginners with no prior experience with programming. Like learning a new natural language, where one must first learn the vocabulary and grammar before understanding the language, a beginner must first learn the syntax and semantics of a new programming language. The syntax defines the combination of symbols that form a valid program, whereas the semantics assign meaning to these valid programs. As learning a programming language is similar to acquiring a natural language, it is unsurprising that this process is much simpler if one already knows a related language. For instance, learning C# is much easier if one knows Java, or picking up English is easier if one is familiar with German. However, many languages used for verification of cyber-physical systems are not related to popular programming languages, both in syntax and semantics. As a result, picking them up and writing models or specifications is challenging.

A popular language used to specify and monitor cyber-physical systems is RTLola. RTLola is a stream-based runtime monitoring framework designed to monitor cyber-physical systems. Using the input data, the monitor produces a verdict on whether the specification is fulfilled. In case the specification is violated, the monitor signals the violation by releasing a trigger such that the CPS can initiate a counter-measure to overcome the unsafe state of the system. RTLola's syntax shares some features similar to the ones of programming languages, such as common arithmetic or boolean operations. However, the biggest difference is the stream-based nature and temporal behavior of the language. The temporal behavior of streams enables the specifier to define when streams are evaluated. This can be done according to a new arrival of input data, or based on a given frequency. Thus, the temporal behavior creates a significant entry barrier for newcomers.

There exist several approaches for teaching programming languages, the most popular one being interactive tutorials. The content of the tutorial is organized into lessons, where each lesson teaches the foundation of key concepts of the language. Typical examples are syntactical rules, how to write a program and execute it, and how to debug a program. After each lesson, a task must be completed to deepen the knowledge of the lesson. These tasks are usually coding tasks that have to be successfully completed before going to the next lesson. This approach requires a lot of commitment as such tutorials can take up to several days to complete.

Another approach is AI-assisted programming, where either the entire program is written by the AI tool or just parts of it. Depending on the complexity of the program, the confidence that it is correct varies. This approach still requires the programmer to be familiar with the syntax and semantics of the programming language to verify the correctness of the program. In a safety-critical setting where the correctness of the program is crucial, this approach is not desirable.

These approaches all cover teaching the text-based representation of the language, which requires teaching the syntax of the language. Since humans are visual learners, we seek a visualization approach in the teaching process. Scratch is the most popular tool that incorporates a visual representation of the language. It features a block-based syntax, meaning that instead of text, it is composed of rectangular blocks that can be connected to each other like Lego pieces. The syntax consists of a set of instructions where each rectangular block represents an instruction. These blocks are color-coded, shaped differently, and categorized based on their context. Scratch abstracts the syntax into the block-based representation. The user is guided to a valid program through the different colors and shapes. Scratch performs well in reducing the cognitive load of considering the correct syntax of the language. In practice, Scratch is used to teach the logic of programming languages with the learning-by-doing process. Users can immediately start writing programs without any prior knowledge by connecting blocks. This keeps the entry barrier low for beginners, as they can build programs and learn the logic of programming languages in an engaging and fun way. As the users progress and familiarize themselves with the logic, they can move on to the text-based representation.

Taking RTLola as an example, we aim to introduce a visualization method that enables users to build specifications intuitively. First, we conducted in Chapter 4 a user study to identify common errors in RTLola specifications. Based on the results of the user study, we propose in Chapter 5 a block-based syntax for RTLola, where the user can build specifications by connecting blocks. The blocks represent the different components of the RTLola language, such as streams, expressions, and types. The blocks are color-coded and shaped differently based on their context. Further, we evaluate in a second user study in Chapter 6 whether the block-based syntax mitigates common errors.

→Chapter 4, p. 15
→Chapter 5, p. 33
→Chapter 6, p. 43

In this thesis, we study the following research questions:

RQ1. What are the common syntactic or semantic errors in RTLola specifications?

RQ2. How can a block-based syntax be effectively adapted to RTLola?

RQ3. Does the block-based syntax approach help to reduce common errors in RTLola specifications?

Chapter 2

Related Work

Writing correct specifications is crucial for the safety of cyber-physical systems. However, writing specifications is a difficult task, especially for beginners without prior experience. Even specifications used in the real world for verifying systems have errors [DFS21]. We report on related work of different approaches to assist users in writing specifications.

The first approach uses AI-assisted program writing. Bucaioni et al. [Buc+24] investigated if ChatGPT can be used to solve programming tasks using Java and C++. They found that while it performs well with easy to medium tasks, the accuracy decreases with more complex tasks. To ensure correctness, the user still needs to verify the code provided by the chatbot. This approach is not suitable for writing specifications in a safety-critical setting since the output is not verified and the user still needs to verify the code provided by the chatbot. Since we aim to create a visualization tool that eases the writing process without professional assistance, this approach may be a good starting point for creating specifications but does not aid the learning process.

Another support in the writing process is interactive web-based IDEs, so-called playgrounds, which allow the user to write a program in the IDE and execute it directly in the browser without the need to install any software. Such web-based IDEs exist for different formal specification languages such as TeSSla, a stream-based specification language [Kal+22] that is used to monitor a system with runtime verification. The TeSSla playground uses a single-window user interface and supports the user by visualizing the output streams, displaying the trace, and the corresponding specification. Another playground for a stream-based specification language is the RTLola playground [FKS23] with a similar user interface displaying the specification, the output, the trace, and a visualization of the streams. For RTLola, the visualizations are more advanced as the dependency graph is created for every specification in the pacing and memory view. The dependency graph assists the specifier in ruling out any false dependencies between the different streams and logical errors in the specification. Despite the first step of visualization, a person with some knowledge of the specification languages is needed to write the specifications.

The third approach involves using visualization tools like Scratch, Simulink, or Lego Mindstorms. Simulink, a popular tool for system-level modeling and simulation, offers a graphical interface for specifying and visualizing dynamic systems [Doc20]. In a study where Simulink was used in an undergraduate electrical engineering course, Duran et al. [Dur+07] found that the student's activity and participation in the course increased and their understanding of the material. Further, the Lego Mindstorms system has been used to visually represent and program robotic behaviors [AK17]. It uses command blocks, which correspond to a robot's direct action. This approach has its main application in a young audience. Lego Mindstorms was one of the first applications to use a block-based syntax.

Scratch [Mal+10] is the most successful and prominent visual programming environment using a block-based syntax. It supports learning how to program by creating programs with colored command blocks. These command blocks are divided into different categories. It uses a single-window user interface to ensure that everything is always visible. There are no error messages, but syntax errors are eliminated since command blocks are shaped so that they only fit together such that no errors are possible, as well as the drag-and-drop feature that does not allow syntactically invalid scripts. Despite eliminating syntax errors, users need to be aware of semantic errors. Similarly to Scratch, this work will use a single-window user interface and a similar concept for the commands. Since RTLola is a stream-based specification language, some different commands will represent streams and the specific timing behavior.

Weintrop and Wilensky conducted a study that showed that a block-based syntax helps novice users learn to write programs in a new language [WW15]. Scratch, being most popular in using a block-based syntax, has shown its success by introducing new students to programming. Ferrer-Mico et al. [FPR12] found that the Scratch programming language helps users gain more intuition about programming in general and increases their learning capabilities.

Scratch is not only used for teaching programming but also has its application in formal verification as a visualization technique for linear temporal logic (LTL). LTL is used to describe temporal properties using atomic propositions, boolean operators, and temporal operators [Pnu77]. The ScratchLTL tool [Gro20] was developed to create LTL formulas using the syntax of Scratch. This approach translates the different LTL components into Scratch commands, enabling the creation of specifications for control systems without focusing on the formalities of the specification language. This enables people with limited knowledge of the details of LTL to create specifications, which are then transformed into an LTL formula.

Chapter 3

Background

This chapter describes the general structure of the RTLola language, focusing on the syntax and type system. We further introduce the block-based syntax and discuss the different block types and their connections. Both sections lay the foundation for the design of SRTLola we present in Chapter 5.

→Chapter 5, p. 33

3.1 RTLola

RTLola is a real-time stream-based specification language used to monitor cyber-physical systems. It is designed to monitor the behavior of a system at runtime. The monitor checks the specifications based on the received input data and signals the system if a violation occurs. In case such a violation happens, the system or an operator can initiate a countermeasure to overcome the unsafe state of the system. Its syntax resembles that of programming languages; however, the biggest difference is the stream-based nature and temporal behavior of the language. The language is a named parameter language, meaning that keywords must be specified explicitly. RTLola provides primitives for common arithmetic and boolean operations and additional RTLola-specific operations, which we will discuss in this section.

This section introduces RTLola intuitively based on the tutorial provided by Baumeister et al. [Bau+25].

3.1.1 Streams

Specifications in RTLola consist of input streams, output streams, and triggers. Input streams represent incoming data, output streams represent the system's behavior, and triggers are used to throw a trigger if a particular property is not satisfied. Before we explore the syntax of RTLola, we will present a simple example written in RTLola.

Example 3.1.1. This example demonstrates a specification that checks if a drone's battery is draining too quickly. We represent the battery level as an input stream and define its value type with `Int64`, which describes the percentage of the battery level. We compute the difference between the current and previous battery levels from the received input data to determine if the battery drains too fast. We use the offset lookup to access the previous value of the battery level. Additionally, for the very first input received for the battery level, we lack a previous value we can use for the offset lookup. Therefore, we provide a default value of 100, assuming that the battery is fully charged at the beginning. Finally, the trigger checks if the battery drain exceeds 50, which indicates that the battery is draining too fast. Since we only need to evaluate the stream when a new value of the battery level arrives, we annotate the output and trigger with `@battery_level`, meaning that every time a new battery level input is received, the output and trigger are evaluated.

```
input battery_level : Int64
output battery_drain : Int64 @battery_level := battery_level.offset(by:
    -1).defaults(to: 100) - battery_level
trigger @battery_level battery_drain > 50 "The battery is draining too fast"
```

△

Input Streams Input streams represent the data received by the monitor. Inputs require a unique name and a value type.

```
input id_in: ValueType
```

Output Streams Output streams represent the behavior of a system. Outputs consist of a unique identifier, the resulting value type. Additionally, the output stream is annotated with a pacing type `@ac`, which indicates when the stream is evaluated. Lastly, the expression computes the result of the stream.

```
output id_out: ValueType @ac := expression
```

There is an alternative definition of output streams that contains three clauses. The spawn and close clause allow for dynamic stream creation, and the eval clause is used to define the expression of the stream. Intuitively, the spawn clause states when the output stream starts to exist, whereas the close clause terminates the stream. In Section 3.1.3.3, the stream's lifecycle is discussed in more detail. Note that the spawn and close clauses are optional and can be omitted, which results in an output stream with just an eval clause. Such a stream is equivalent to the output stream definition above.

➔Section 3.1.3.3, p. 10

```
output id_out : Type
 spawn @pacing_spawn when condition_spawn
 eval @pacing_eval when condition_eval with expression
 close @pacing_close when condition_close
```

Trigger Streams Trigger streams encode when the system enters a critical state. Triggers are declared with an expression stating a boolean constraint. Whenever a trigger evaluates to true, the trigger is released, and the trigger message is displayed. Further, triggers are annotated with a pacing type that indicates when the trigger is evaluated.

```
trigger @ac expression "trigger_message"
```

3.1.2 RTLola Expressions

In RTLola, expressions are used to compute the value of an output stream. Expressions can be arithmetic operations, boolean expressions, or specific expressions for RTLola that involve stream lookups. We introduce the different stream lookups in RTLola in the following.

Synchronous Lookup The synchronous lookup is used to access the current value of a stream by stating the stream name as follows:

```
s_ref
```

Offset Lookup Offset lookups access past values of a stream. The n^{th} previous value of a stream is used by stating the offset number n. In case the value does not exist yet, a default value must be provided. The offset lookup is used as follows:

```
s_ref.offset(by: -n)
```

Descrete Sliding Window Lookup An aggregation over a discrete stream window `s_ref` is computed with a sequence of n values using a function `f`. Intuitively, this is very similar to the offset lookup. However, the difference is that the sliding window takes the sequence over the last n values to compute the aggregation over these values and does not take an offset value. These n values serve as the sliding window. The function `f` is an aggregation function that is applied to the values of the stream. Depending on the aggregation function, a default value must be provided. The syntax of the sliding window lookup is as follows:

```
s_ref.aggregate(over_discrete: n, using: f)
```

Sliding Window Lookup An aggregation of a stream `s_ref` over a real-time window `t` using a function `f`. The values in the time window are accessed asynchronously. The function `f` is an aggregation function that is applied to the values of the stream. Depending on the aggregation function, a default value must be provided. The syntax of the sliding window lookup is described as follows:

```
s_ref.aggregate(over: t, using: f)
```

Hold Lookup Hold lookups are used to asynchronously access the latest value of a stream. Intuitively, this lookup is usually used when a synchronous lookup is not possible. A default value must be provided in case the value does not exist yet. The hold lookup is used as follows:

```
s_ref.hold()
```

Default Expression Since some lookups can fail if the value does not exist yet, a default expression must be provided. This expression is used as a fallback value in case the lookup fails. The default expression is used as follows:

```
expr.defaults(to: expr)
```

Example 3.1.2. The following example demonstrates the usage of different stream lookups. The specification models a heart patient.

```
input heart_rate : Float64
input blood_pressure : Float64

output avg_heart_rate : Float64 @1min := heart_rate.aggregate(over: 1min, using:
    avg).defaults(to: 0.0)
output hr_diff : Float64 @1min := avg_heart_rate - avg_heart_rate.offset(by:
    -1).defaults(to: 0.0)
output hr_bp: Bool @blood_pressure := (hr_diff.hold().defaults(to: 0.0) > 20.0) &&
    (blood_pressure > 120.0)
trigger @1min hr_diff > 20.0 "The average heart differs too much"
```

The specification has two input streams: `heart_rate` and `heart_rate`.

The output stream `avg_heart_rate` computes the average heart rate over a one-minute window. Indicated by the pacing type `@1min`, the stream is evaluated every minute. To collect all values of `heart_rate` over one minute, the sliding window lookup is used with the aggregation window of one minute and the aggregation function `avg`. The default value is set to 0.0 in case no value is available.

The output stream `hr_diff` computes the difference between the current and previous average heart rate. To access the previous value of the average heart rate, the offset lookup is used. To ensure that the offset lookup does not fail, a default value of 0.0 is provided.

The output stream `hr_bp` checks if the difference between the current and previous average heart rate is greater than 20 and the blood pressure is greater than 120.

The trigger stream checks if `hr_diff` is greater than 20. △

3.1.3 Type System

In RTLola, types are represented by a pair consisting of the value type and the pacing type of a stream.

3.1.3.1 Value Type

The value type defines the size and the domain of the resulting value of a stream. Common programming languages like Java or C# also use value types such as `int`, `String`, `float`, or `boolean`. In RTLola, value types range from boolean values, signed and unsigned integers, to floating-point numbers. Except for the boolean value type, all value types are annotated with the corresponding bit length.

Example 3.1.3. The value type of the output stream `output s : Int64 := a` is `Int64`. △

3.1.3.2 Pacing Type

The pacing type of a stream describes its timing behavior, particularly when the stream is evaluated. It can be either event-based or periodic and is annotated by the `@` keyword. Pacing types include a timeline of a stream. The timeline includes the time points when the stream is evaluated and when the stream starts to exist or is closed.

Event-based Streams Event-based streams are evaluated depending on an event corresponding to the arrival of new values of input streams. Event-based types contain an activation condition, a positive boolean formula that refers to input streams.

Recall Example 3.1.1 where `battery_drain` computes the difference between the current and previous battery level. The stream is evaluated whenever a new value of the input stream `battery_level` arrives, resulting in the event-based type `@battery_level`. →Example 3.1.1, p. 5

Periodic Streams So far, the evaluation of streams was merely dependent on the arrival of new values of input streams. But we can also evaluate streams periodically. To indicate that a stream is evaluated periodically, we state the period of the stream with a time unit. The period is the time interval between two evaluations of the stream. The timeline of a periodic stream is independent of the arrival of new values of input streams. Therefore, to do stream lookups between periodic and event-based streams, we need to use the hold lookup.

Recall Example 3.1.2 where the output streams `avg_heart_rate` and `hr_diff` are periodic streams that are evaluated every minute. The pacing type of these streams is `@1min`. Since `avg_heart_rate` produces a value every minute, we can access `avg_heart_rate` synchronously. However, `hr_bp` is evaluated with an event-based type of `@blood_pressure`. Since we cannot ensure that `blood_pressure` produces a value every minute, we need to use the hold lookup to access the periodic output stream of `hr_diff`. →Example 3.1.2, p. 8

Semantic Type On top of the event-based and periodic types, we can also define a semantic type. It is introduced by the `when` keyword followed by an RTLola expression. The semantic type further refines the timeline of a stream. With the semantic type we allow restrictions to the time points of the timeline of a stream. If the semantic type is not satisfied, the time point is skipped in the timeline.

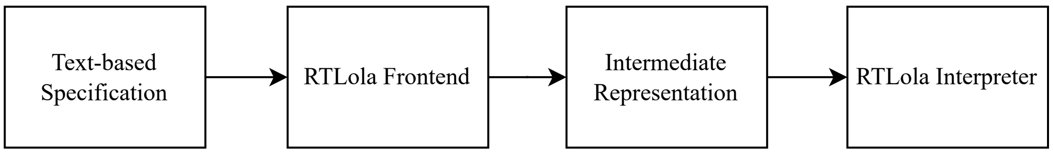


Figure 3.1: Overview of the RTLola framework

3.1.3.3 Stream Lifecycle

In RTLola, streams can be created dynamically. This means we can define when a stream starts to exist and when it is closed. Recall the definition of an output stream with the spawn, eval, and close clause. The spawn clause defines when the stream starts to exist, the eval clause defines the expression of the stream, and the close clause defines when the stream is closed. Consider the scenario where a stream is spawned according to an event-based type, and the eval clause is evaluated periodically. Then, the timeline either aligns with the global-periodic clock or does not align and is evaluated according to the local clock. The local clock starts when the stream is spawned and stops when the stream is closed.

3.1.4 Framework

The RTLola framework consists of the RTLola frontend and the interpreter. First, the specification is parsed and analyzed by the frontend. The analysis consists of three levels. The first level is the syntactical analysis, which checks the specification for syntax errors. The second level is the value type analysis, which evaluates the specification and checks if value type errors occur. The pacing type analysis is the third level and examines whether the specification contains pacing type errors. These analyses form the base for the evaluation of both user studies discussed in Chapter 4 and Chapter 6.

→Chapter 4, p. 15
→Chapter 6, p. 43

After the frontend has analyzed the specification, an intermediate representation is produced that is used by the RTLola interpreter. The interpreter evaluates the specification based on the incoming data.

3.2 Block-Based Syntax

This section introduces the block-based syntax inspired by Scratch based on the Google Blockly library we will use in this thesis. We discuss which block types exist and how we build a program from these blocks.

The block-based syntax provides a way of connecting blocks such that syntactically incorrect programs are not allowed by design. The block-based syntax consists of block types divided into three categories: outer blocks, statement blocks, and input blocks. We will explain the properties of these categories and how they can be connected.

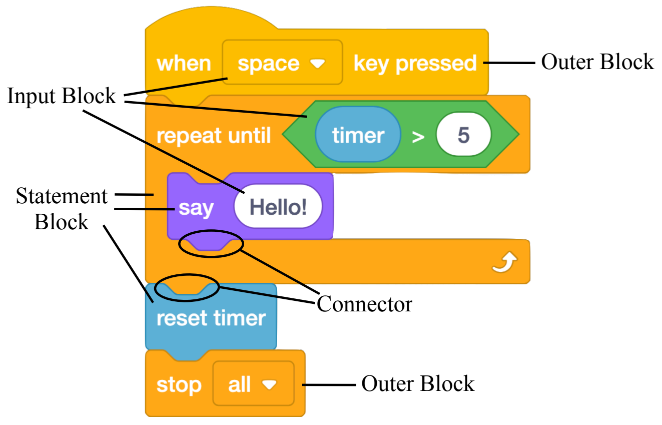


Figure 3.2: Example Program in Scratch

To see what a program in a block-based syntax looks like, we will start with an example program in Scratch.

Example 3.2.1. Figure 3.2 shows an example program in Scratch. This program lets the Scratch cat say "Hello!" depending on a timer after starting the program. The overall structure of the program reminds us of a program written in a programming language. →Figure 3.2, p. 11

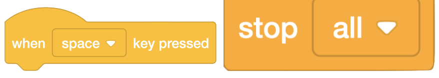
The top block specifies that the program starts when the user presses the space key. Then, we introduce a repeat statement that repeats the say hello statement until the timer exceeds 5 seconds. After the timer is greater than 5 seconds, the program resets the timer to 0 and terminates. △

3.2.1 Outer Blocks

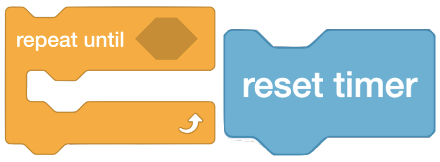
TThe first category is the starting point for any program we want to create with the block-based syntax. Figure 3.3a shows two outer blocks with its connector. As we have seen in Example 3.2.1, the Scratch program starts with an outer block and ends with an outer block. These blocks build the frame for the program. In our context, a *program* is the collection of block groupings that start with an outer block and all blocks connected to them. →Figure 3.3a, p. 12 →Example 3.2.1, p. 11

3.2.2 Statement Blocks

Statement blocks are the second category of blocks. They can be either connected to each other or to outer blocks. Figure 3.3b demonstrates that statement blocks can be nested, meaning it is possible to have a statement input with several blank spaces for other statement inputs. →Figure 3.3b, p. 12



(a) Outer Blocks with Connectors



(b) Statement Blocks with Connectors

Figure 3.3: Connectors

3.2.3 Connectors

Outer blocks and statement blocks have connectors that allow us to connect them to each other. Outer blocks have only one connector, which makes them either to the start or the end of a program, whereas statements have multiple connectors. Blocks can have blank spaces in the middle, and each blank space has two connectors. The blank spaces are filled with statement blocks. *Connectors* enable us to connect blocks like puzzle pieces. The different connectors allow us to rule out certain combinations of blocks. Blocks that cannot be connected have different connector shapes. The block-based syntax of Scratch does not use different connector shapes for its syntax. However, in a more complex language such as RTLola, this feature allows us to differentiate between multiple categories of expressions.

3.2.4 Input Blocks

→Figure 3.4, p. 13

Input blocks have a matching placeholder with the same shape as the input block, which is found throughout all block types. Figure 3.4 shows two input blocks where the right one is a nested input block. There are several types of input blocks: text blocks, dropdown blocks, and variable blocks.

3.2.4.1 Dropdown Blocks

→Figure 3.2, p. 11

Figure 3.2 has a dropdown block in the first outer block where we can choose a key to start the program. Dropdown blocks provide a predefined selection of options. When you have more than five groups of options, this is the preferred block type to display all options without introducing a new shape for each group.

3.2.4.2 Text Blocks

→Figure 3.4, p. 13

Figure 3.4 shows a text block to define to which number we should compare the timer. For text blocks, the user has to provide their input for the given text input block. For a block-based syntax, we want to avoid this type of block as much as possible since we cannot control the user's input and cannot guide the user to provide the correct input for the text block. Whereas text blocks are a suitable option for inputs such as numbers or error messages, text blocks are an unfitting option for case-sensitive inputs.

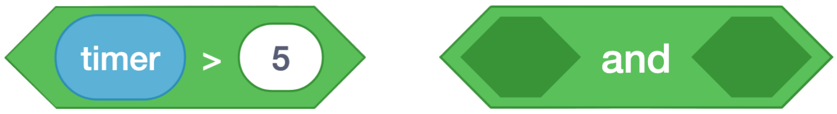


Figure 3.4: Input Blocks

3.2.4.3 Variable Blocks

The timer input block is a variable block in Figure 3.4. Variable blocks are input blocks created before being used in a program. They provide a way of using a block multiple times. Variable blocks correspond to defining a variable in a program, hence the name.

→Figure 3.4, p. 13

Chapter 4

Finding Common Errors in RTLola Specifications

This chapter introduces the methodology and structures of the first user study. This user study answers the first research question of this thesis: *What are the common syntactic or semantic errors in RTLola specifications?*. Further, we explain all tasks of the study in more detail, including the teaching goal, exercise description, and sample solution. In Section 4.6, we evaluate the user study and categorize the common errors. ➔Section 4.6, p. 28

4.1 Study Design

This user study aims to identify the difficulties of learning the RTLola specification language and common errors people make while writing specifications. We provide a tutorial that covers the basic syntax and semantics of RTLola explained in Section 3.1. ➔Section 3.1, p. 5 During the user study, the participants wrote RTLola specifications. These, combined with the matching errors, were gathered during the study and stored for further evaluation. We conducted the user study with ten participants (eight male and two female, with an average age of 24). All participants have a background in computer science, either as undergraduates or as PhD students in computer science.

4.2 Apparatus

An RTLola tutorial was provided for the user study and integrated into the RTLola playground[FKS23]. The tutorial guides the participants through the user study by presenting the different components of RTLola and describing the different tasks they should solve and submit. After each run of their written RTLola specification in the playground, the specification is stored for future evaluation. In the tutorial, the specification editor does not provide any live suggestions or error messages from the RTLola front end. Thus, it serves as a plain text editor. Only after running the specification are error messages displayed in the console to help the participants correct their specifications. Their expertise level with RTLola ranges from more advanced users to users who have never seen the language.

4.3 Procedure

Each participant conducted the user study alone in a room with the researcher, ensuring a controlled environment free from distractions. We asked about demographics and the participants' prior experience with RTLola. At the beginning of the study, participants were provided with a user consent form, which included consent for audio recording. The researcher then took the time to explain the study procedures in detail, ensuring that all participants were comfortable doing the tutorial. The participants completed six chapters containing five tasks throughout the study, explained in more detail in Section 4.5. Feedback was provided through error messages displayed in the console after each specification run. The researcher offered additional assistance if the participants encountered difficulties correcting the specifications.

→ Section 4.5, p. 18

4.4 Data Evaluation

To evaluate the user study in more detail, we collected the written RTLola specifications the participants provided during the tutorial. For additional information we stored the given trace as well as the chapter the participants were in when they wrote the specification. The data was stored as JSON files on the hard drive of the researcher. To ensure the privacy of the participants the data was anonymized and the participants were assigned a random number.

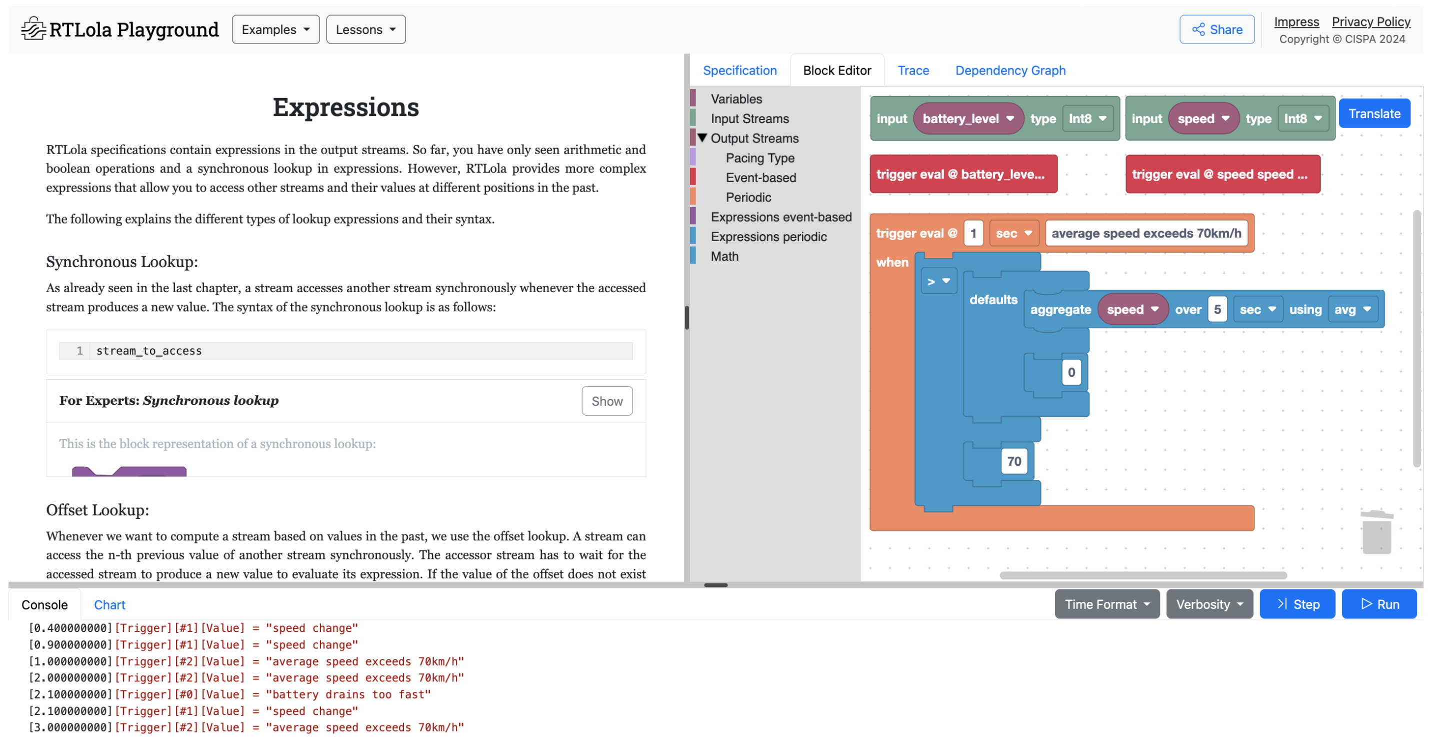
After the user study, each file was annotated with the errors the RTLola frontend displayed in the console for the corresponding specification. According to the analysis level where the error occurred, the errors were categorized into value type errors, pacing type errors, and syntax errors. We evaluated all specifications manually to ensure that the errors were correctly categorized and to provide further information on why the error occurred under the key `reasons`.

→ Figure 4.2, p. 18

Example 4.4.1. Figure 4.2 shows an example of an annotated file of a participant after evaluation. The file contains the errors of the given specification if they exist. The error contains the error message, the reasons for the error, the error type, and the severity of the error. The file also contains the chapter for which the participant solved the task, the specification that was submitted, and the trace.

△

Figure 4.1: RTLola Playground Tutorial



Expressions

RTLola specifications contain expressions in the output streams. So far, you have only seen arithmetic and boolean operations and a synchronous lookup in expressions. However, RTLola provides more complex expressions that allow you to access other streams and their values at different positions in the past.

The following explains the different types of lookup expressions and their syntax.

Synchronous Lookup:

As already seen in the last chapter, a stream accesses another stream synchronously whenever the accessed stream produces a new value. The syntax of the synchronous lookup is as follows:

```
stream_to_access
```

For Experts: *Synchronous lookup*

This is the block representation of a synchronous lookup:

Offset Lookup:

Whenever we want to compute a stream based on values in the past, we use the offset lookup. A stream can access the n-th previous value of another stream synchronously. The accessor stream has to wait for the accessed stream to produce a new value to evaluate its expression. If the value of the offset does not exist

```
[0.400000000][Trigger][#1][Value] = "speed change"
[0.900000000][Trigger][#1][Value] = "speed change"
[1.000000000][Trigger][#2][Value] = "average speed exceeds 70km/h"
[2.000000000][Trigger][#2][Value] = "average speed exceeds 70km/h"
[2.100000000][Trigger][#0][Value] = "battery drains too fast"
[2.100000000][Trigger][#1][Value] = "speed change"
[3.000000000][Trigger][#2][Value] = "average speed exceeds 70km/h"
```

```
{
"error": [
{
   "message": "In value type analysis:\nFound incompatible types: Option<?> and
       Numeric",
   "notes": [],
   "reasons": "whenever option expected, a defaults was forgotten either in offset,
       hold, or aggregate",
   "error_type": "value_type_error",
   "severity": "Error"
}
],
"route": "/playground/lessons/tutorial/tut_03_expressions",
"spec": "input speed: Float64\ninput battery_level: Float64\n\noutput speed_change
    := speed - speed.offset(by: -1).defaults(to: 0.0)\ntrigger speed_change > 14.0
    \"The speed change of drone was too drastically\"\n\noutput battery_drain :=
    battery_level - battery_level.offset(by: -1)\ntrigger battery_drain > 50 \"the
    battery drains too fast\"\n\noutput avg_speed := speed.aggregate(over_discrete:
    5, using: average)\ntrigger avg_speed > 70 \"avg speed is too fast\"",
"trace": "speed,battery_level,time\n0.0,100.0,0.1\n10.0,95.0,0.2\n30.0,95.0,0.4\n
    35.0,93.0,0.5\n30.0,92.0,0.9\n25.0,91.0,1.1\n55.0,85.0,2.1\n10.0,65.0,3.1"
}
```

Figure 4.2: Annotated file of a participant after evaluation

Remark 4.4.1. *The RTLola Frontend panicked in several specifications in the pacing type analysis. These specifications were either correct or contained an offset lookup with a future offset, e.g.,* `altitude.offset(by: 1).defaults(to: 0)` *instead of* `altitude.offset(by: -1).defaults(to: 0)`*. The RTLola frontend does not support future offsets and panics in these cases. We manually checked all the specifications, which caused a panic. In the case where the specification is correct, we annotated these files with an empty* `"spec"`*. The specifications that contained future offsets were corrected to evaluate the remaining errors. In all annotated files, we kept the original specification.*

4.5 Tasks

To complete the tutorial, the participants must write five RTLola specifications. Each task has the following structure: The definition of the task is given in natural language, containing the information needed to write the RTLola specification. All tasks have an input file provided, and some have an additional template to guide the participants through the writing process.

The five tasks aim to gradually introduce more complex concepts of RTLola. In this tutorial, we will explore three scenarios. The first scenario models a drone checking its altitude and whether it is in the landing zone or not. In the fourth task, we extend the first scenario by adding the GPS signal strength of the drone to the monitor. The second scenario also models a drone but checksks the speed and battery level. Lastly, in the third scenario, we create a monitor for a heart patient, where the heart rate and blood pressure are tracked.

In the following, we explain all tasks and the teaching goal. Then, we specify what the specification has to monitor by a description and give a sample solution, which is usually divided into input streams and output/trigger stream pairs that match the task description.

4.5.1 Task 1

The first task covers the first scenario with the altitude and landing zone as input streams.

Teaching Goal This task aims to introduce the participants to the basic syntax of RTLola and familiarize them with the playground and how to run a specification with a given trace. The main focus of this task is the expressions of the output and trigger streams. We explained the stream-based nature of the language and the different types of streams, including the syntax of input and output streams and how they are used in a specification. Further, we showed simple expressions, like boolean and arithmetic expressions, and the first RTLola-specific expression, the synchronous lookup.

Exercise To help participants with the syntax of the language, we provided a template with placeholders for the expressions in the output and trigger streams. They are denoted as `expression` and `trigger_expr` as displayed in Figure 4.4. ➔Figure 4.4, p. 20

	ID	Description
Inputs	`altitude`	Current altitude
	`landing_zone`	True if the system is in a landing zone
Outputs	`altitude_below_30`	Altitude is below 30
	`too_low_not_in_landing_zone`	Altitude is below 30 and not in landing zone

Solution Specification 1 In Section 4.5.1, the input stream `altitude` is given as an integer, and `landing_zone` is given as a boolean value. We compute the first output by comparing ➔Section 4.5.1, p. 20

```
input altitude : Int64
input landing_zone : Bool

output altitude_below_30 := expression
trigger trigger_expr "The drone is too low"

output too_low_not_in_landing_zone := expression
trigger trigger_expr "The drone is too low and cannot land"
```

Figure 4.4: Template of Task 1

```
input altitude : Int64
input landing_zone : Bool

output altitude_below_30 : Bool := altitude < 30
trigger altitude_below_30 "The drone is too low"

output too_low_not_in_landing_zone : Bool := altitude_below_30 && ¬ landing_zone
trigger too_low_not_in_landing_zone "The drone is too low and cannot land"
```

Listing 4.1: Sample Solution Specification 1

the altitude with a threshold of 30. Then, we define a trigger that is thrown if the output evaluates to true with the trigger message `"The drone is too low"`.

The second output is computed by checking the boolean condition, which evaluates to true if the drone is not in its landing zone and is below 30 meters `altitude_below_30 && ¬ landing_zone`. If the output evaluates to true, the trigger with the trigger message `"The drone is too low and cannot land"` is thrown.

4.5.2 Task 2

The second task covers the second scenario with the speed and battery level as input streams. We switch the scenarios to solidify the syntax of RTLola in a different context and prevent participants from copying their solution from the previous task and reusing it.

Teaching Goal The second task builds on the syntax and basic expressions introduced in the first one. We introduce the participants to the first part of RTLola expressions, such as the synchronous lookup, the offset lookup, and the aggregate expression over a discrete window. The participants learn how to use these expressions, when to use which expression, and for which expression a default value is needed.

```
input speed: Float64
input battery_level: Float64

output speed_change := expression
trigger trigger_expr "The speed change of drone was too drastically"
```

Figure 4.6: Template of Task 2

Exercise The second task switches the scenario to change the thought process of the participants. In this task, a template, shown in Figure 4.6, was also provided, but only the input streams and one output trigger pair template are given. This way, the participants must write two pairs of output trigger declarations from scratch and become more familiar with the RTLola language.

→Figure 4.6, p. 21

We still offer some guidance for the participants to remember the syntax of RTLola and, therefore, also provide a template shown in Figure 4.6. It covers the input streams and the first output stream with its corresponding trigger where the expressions are replaced with `expression` and `trigger_expr` similar to Section 4.5.1. However, we also encourage the participants to write the remaining two properties of the specification independently.

→Figure 4.6, p. 21
→Section 4.5.1, p. 19

	ID	**Description**
Inputs	`speed`	Current speed of the system in km/h
	`battery_level`	Current battery level
Outputs	`speed_change`	System changes speed by more than 14 km/h between the last two speed inputs
	`battery_drain_rate`	Rate at which the battery level drains is greater than 50 percent between the latest two battery level inputs
	`avg_speed`	Average speed over the last five values exceeds 70.0 km/h

Solution Specification 2 To model the first trigger, we create an output stream `speed_change` that computes the difference between the two latest speed values by accessing the previous value with an offset lookup by one and the current value with a synchronous lookup. Also, we provide the default value of 0 for the offset lookup. Then, we declare a trigger that checks if `speed_change` is greater than 14.0.

For the second trigger, we create an output stream `battery_drain_rate` that computes the difference between the latest two values of `battery_level` by accessing the previous value with an offset lookup by one and the current value with a synchronous lookup. Also, we provide the default value of 100 for the offset lookup, assuming that the drone is at full battery level. Then we declare a trigger that checks the condition (`battery_drain_rate>50.0`)

We use an aggregation expression for the third trigger by setting the sliding window to 5 and using the average function. Then, we throw a trigger if (`avg_speed > 70.0`).

```
input speed: Float64
input battery_level: Float64

output speed_change := speed - speed.offset(by: -1).defaults(to: speed)
trigger speed_change > 14.0 "The speed change of the drone was too drastically"

output battery_drain_rate := battery_level.offset(by: -1).defaults(to: 100.0) -
    battery_level
trigger battery_drain_rate > 50.0 "The battery level dropped by 50 percent"

output avg_speed := speed.aggregate(over_discrete: 5, using: avg).defaults(to: 0.0)
trigger avg_speed > 70.0 "Average speed over the last five values is above 70"
```

Listing 4.2: Sample Solution Specification 2

4.5.3 Task 3

The third task switches back to the first scenario. Since the participants are familiar with some parts of the specification, it should give them confidence to write and extend the specification.

Teaching Goal This task introduces the second part of RTLola expressions, such as the hold lookup and sliding windows over a continuous time window. In that context, the RTLola types were introduced, especially the pacing types. The emphasis of this task is to show when to use periodic pacing types and how to do a lookup to an event-based stream from a periodic stream and vice versa.

Exercise This exercise extends the exercise of the first task and contains output trigger pairs that are identical or similar to the pairs from the first task.

	ID	Description
Inputs	`altitude`	Current altitude

	ID	Description
	`landing_zone`	True if the system is in a landing zone
Outputs	`altitude_below_30`	Altitude is below 30
	`altitude_diff`	Every time the drone is flying too low, check the difference between the latest two values of altitude and check if the difference is greater than 30
	`avg_altitude`	Check if the average altitude over one minute is greater than 50 meters
	`too_low_not_in_landing_zone`	Every time altitude is below 30, check if the system is not in landing zone

Solution Specification 3 The first trigger is computed as in the specification of the first task.

In the output stream `altitude_diff`, the semantic filter has to be applied with the condition `altitude_blow_30` to compute the difference between the past two values.

The output stream `avg_altitude` computes the average altitude with a sliding window lookup over one minute. The output stream is annotated with a frequency of 3 minutes, which means that the output stream is evaluated every three minutes.

The semantic filter in the last output stream is applied as in `altitude_diff`. For the trigger of `too_low_not_in_landing_zone` and the `altitude_diff`, a hold lookup to access the values of the output streams is necessary. Because of the semantic filter, a new value is not produced every time the activation condition is fulfilled.

```
input altitude : Int64
input landing_zone : Bool

output altitude_below_30 : Bool := altitude < 30
trigger altitude_below_30 "The drone is too low"

output altitude_diff : Int64 eval when altitude_below_30 with altitude -
    altitude.offset(by: -1).defaults(to: altitude)
trigger @altitude altitude_diff.hold().defaults(to: 0) > 30 "Trigger"

output avg_altitude @3min := altitude.aggregate(over: 1min, using: avg).defaults(to:
    0)
trigger avg_altitude > 50 "Average altitude over the last minute is too high"

output too_low_not_in_landing_zone : Bool eval when altitude_below_30 with ¬
    landing_zone
```

```
trigger @(altitude && landing_zone) too_low_not_in_landing_zone.hold().defaults(to:
    false) "The drone is too low and cannot land"
```

Listing 4.3: Sample Solution Specification 3

4.5.4 Task 4

Teaching Goal The fourth task introduces the spawn, eval, close causes, and relative clock of streams. Its emphasis is on showing how dynamic stream creation works and how to use it.

Exercise This exercise extends the specification from the third task with the additional input stream `gps_signal_strength`. The output and trigger pairs, except for `signal__strength`, from this exercise extend the pairs from the previous exercise but add spawn and close clauses to them. Further, to avoid the participants purely annotating their previous solution, we provided a template with only the three input streams.

	ID	Description
Inputs	`altitude`	Current altitude
	`landing_zone`	True if the system is in a landing zone
	`gps_signal_strength`	GPS signal strength represented as an integer
Outputs	`altitude_below_30`	Altitude is below 30
	`avg_altitude`	Every time the drone is flying too low, we want to check every 3 minutes if the average altitude is also below 30 meters
	`signal_strength`	Every time the GPS signal is less than five, we want to check if the last value of the GPS signal strength was already below ten and not check the condition if the signal strength is above 20 again
	`too_low_not_in_landing_zone`	Every time the drone is flying too low, we want to check if the drone is not in its landing zone and not check the condition anymore if the drone is above 40 meters again. If the drone is not in its landing zone, throw a trigger

Solution Specification 4 In the output stream `avg_altitude`, the difficulty is to see what the spawn condition is; in this case, `altitude_below_30`, and the frequency of 3 minutes must be annotated.

The output stream `signal_strength` needs the spawn condition `signal_below_5` and closes the stream when the signal is above 20.

Overall, the most difficult part for the participants was to use a hold lookup to the dynamic streams for the trigger streams. Therefore, the pacing type has to be stated explicitly.

```
input altitude : UInt64
input landing_zone : Bool
input gps_signal_strength : UInt64

output altitude_below_30 : Bool := altitude < 30
trigger altitude_below_30 "The drone is too low"

output avg_altitude : UInt64
   spawn when altitude_below_30
   eval @3min when true with altitude.aggregate(over: 1min, using: avg).defaults(to:
       0)
trigger @(altitude) avg_altitude.hold().defaults(to: 30) < 30 "The average altitude
    is also too low"

output signal_strength : UInt64
   spawn when gps_signal_strength<5
   eval when true with gps_signal_strength.offset(by: -1).defaults(to: 0)
   close when gps_signal_strength > 20
trigger @(gps_signal_strength) signal_strength.hold().defaults(to: 0) < 10 "GPS
    signal is not strong enough to capture the drone"

output too_low_not_in_landing_zone : Bool
   spawn when altitude_below_30
   eval when true with ¬ landing_zone
   close when (altitude > 40)
trigger @(altitude && landing_zone) too_low_not_in_landing_zone.hold().defaults(to:
    false) "The drone is too low and cannot land"
```

Listing 4.4: Sample Solution Specification 4

4.5.5 Task 5

Teaching Goal The fifth task puts everything together and recaps the previous tasks. The participants have to write a specification with no template provided. The goal is to see if the participants can remember the syntax and semantics of RTLola and apply them to a more complex specification.

Exercise The fifth task models a heart patient based on the blood pressure and heart rate. The participants have to write a specification with the following properties:

	ID	**Description**
Inputs	`blood_pressure`	The patient's blood pressure
	`heart_rate`	The patient's heart rate given in beats per minute
Outputs	`bp_high`	The blood pressure is too high when it is above 120
	`bp_high_count`	Check every 30s if the blood pressure is too high more than five times over a minute and if so, we want to throw a trigger
	`hypertension`	Compute every 30s when the blood pressure is above 160. This computation is only necessary when the patient has high blood pressure. We filter this computation by checking if the blood pressure was too high more than 5 times over the last minute. If this count is less than three, we don't need to check this property anymore. Then, we want to release a trigger if the patient is indeed in hypertension.
	`avg_heart_rate`	Compute the average heart rate over one minute
	`hr_diff`	Compare the past two computed average heart rates over one minute and release a trigger if the difference is above 20. Since we do not have enough resources to check this property every 30 seconds, a longer period needs to be provided.
	`hypertension_and_avg_hr`	Combine the two properties, hypertension and average heart rate. If the patient has hypertension and the average heart rate is over 100, we want to throw a trigger as a response

Solution Specification 5 While writing the specification, the participants encountered some difficulties. For the output stream `high_bp_count`, it was counterintuitive to aggregate over the output stream `bp_high` to count the times the blood pressure is too high.

Since this specification contains several dynamic streams, the participants had to pay attention to using the correct conditions. When accessing a dynamic stream in a regular stream, it must be accessed with a hold lookup, or the regular stream must be transformed into a dynamic stream, fulfilling the conditions of the accessed stream. In the sample solution, `hypertension` is a dynamic stream that has to be accessed with a hold lookup in the output stream `hypertension_and_avg_heart_rate`. Similarly, in the event-based output stream `hypertension_and_avg_heart_rate`, the periodic stream `avg_heart_rate` has to be accessed with a hold lookup.

In the case of only hold lookups in the expression, the pacing type must be stated explicitly, such as in the trigger of the output streams `hypertension_and_avg_heart_rate` and `hypertension`.

```
input blood_pressure : Float64
input heart_rate : Float64

// high bp
output bp_high : Bool @blood_pressure := blood_pressure > 120.0
output bp_high_count : UInt64 @30s := bp_high.aggregate(over: 1min, using: count)
trigger bp_high_count > 5 "The blood pressure was too high the last 5 times over a
    minute"
// we only monitor the systolic blood pressure

// hypertension
output hypertension : Bool spawn when bp_high eval @30s when bp_high_count > 5 with
    blood_pressure.hold().defaults(to: 80.0) > 160.0 close when bp_high_count < 3
trigger @(blood_pressure) hypertension.hold().defaults(to: false) "Patient's blood
    pressure is critically high"

// heart rate
output avg_heart_rate : Float64 @1min := heart_rate.aggregate(over: 1min, using:
    avg).defaults(to: 0.0)
output hr_diff : Float64 @1min := avg_heart_rate - avg_heart_rate.offset(by:
    -1).defaults(to: 0.0)
trigger hr_diff > 20.0 "The average heart differs too much"

//hypertension and heart rate
output hypertension_and_avg_heart_rate spawn when bp_high eval @60s when
    bp_high_count > 5 with hypertension.hold().defaults(to: false) &&
    (avg_heart_rate.hold().defaults(to: 0.0) > 100.0)
```

```
trigger @(blood_pressure && heart_rate)
    hypertension_and_avg_heart_rate.hold().defaults(to: false) "High blood pressure
    and high average heart rate detected"
```

Listing 4.5: Sample Solution Specification 5

4.6 Results and Discussion

We have identified several common errors while writing RTLola specifications in this user study. These errors can be divided into different categories: value type errors, pacing type errors, and syntax errors. We will explain the errors in more detail in the following. The examples shown are code snippets of specifications the participants wrote during the user study.

4.6.1 Value Type Errors

In RTLola, we enforce operations with the same value type, i.e., we cannot add an integer value to a float value. Therefore, the user has to be aware of the value types of all the streams and which value types the operations allow. Further, a default expression must be provided for expressions such as the hold lookup, sliding window lookup, and offset lookup.

Incorrect Value Type Some participants have given incorrect value types that either do not exist in RTLola or are misspelled. For example, the following input stream was defined: `input heart_rate : float`. Although floating point numbers exist in RTLola, they have to be specified as `input heart_rate : Float64`

Omitted Default Expression One of the most common errors was the omitted default expression. A default expression must be provided for hold lookups, offset lookups, and sliding window lookups. If the default expression is omitted, it produces an error in the value type analysis since the type of these expressions without a default expression is an optional type. This error is solved by a default value with the correct value type.

Example 4.6.1. This code snippet should compute the difference between the last two value inputs of `battery_level`. Often, the solution provided by the participants was: `output battery_drain := battery_level - battery_level.offset(by: -1)`.

The correct solution for this part is to add a default value for `battery_level`: `output battery_drain := battery_level - battery_level.offset(by: -1).defaults(to: 100.0)` △

Operation with Different Value Types RTLola does not support typecasting. Therefore, operations within the expressions must be done with inputs and values of the same value type. Several errors occurred because integer values were mixed up with float values since participants assumed RTLola could cast these values. However, there were also some errors that were more severe and not that simple to detect, as the following example shows.

Example 4.6.2. In this code snippet, we count the times the blood pressure is too high and throw a trigger every time that count exceeds five. In the following, a participant provides an incorrect default value for the output stream of `high_BP`. Instead of `0` the default value `false` was given.

```
input blood_pressure : Float64
output high_BP
   eval @30s with blood_pressure_too_high.aggregate(over: 1min, using: count)
trigger @(blood_pressure) high_BP.hold().defaults(to:false) > 5 "Blood pressure too
     high"
```

△

4.6.2 Pacing Type Errors

The pacing type is an essential aspect of the RTLola language. Usually, the pacing type can be derived from the stream accesses used in the expression. However, if the pacing type cannot be derived, e.g., if the expression contains purely hold lookups, the user has to state the pacing type explicitly. This leads to several errors during the user study.

Malformed Pacing Type with Output Streams The event-based pacing type must only contain input streams. Several participants provided an activation condition with an output stream.

Example 4.6.3. In this code snippet, the output stream `prop3` is spawned if the altitude is too low and closed if it is above 40. The pacing type `@(landing_zone && too_low)` contains the output stream `too_low` which has to be omitted. So, the correct pacing type is `@landing_zone`.

```
input altitude : UInt64
input landing_zone : Bool
output too_low := altitude < 30
output prop3
spawn when too_low
   eval @(landing_zone && too_low) with landing_zone
close when altitude > 40
```

△

Omitted Pacing Type Hold Lookup The pacing type cannot be derived if the expression contains only hold lookups. Since most participants relied on the type inference, they forgot to provide a pacing type in these cases.

Example 4.6.4. In this code example, the trigger stream accesses the output stream `too_low_and_not_in_landing_zone` with a hold lookup. Therefore, the required pacing type was omitted. To correct the specification, the pacing type `@(altitude && landing_zone)` must be stated in the trigger.

```
input altitude : Int64
input landing_zone : Bool
output drone_level_too_low := altitude < 30
output too_low_and_not_in_landing_zone
   eval @landing_zone when drone_level_too_low with landing_zone
trigger too_low_and_not_in_landing_zone.hold().defaults(to: false) "Drone is flying
     too low and is also not in its landing zone."
```

△

Frequency of Periodic Pacing Type If a periodic stream accesses another periodic one, the annotated frequency must be the greatest common divider (GCD) of the accessed stream.

Example 4.6.5. In this code example, `avg_HR` computes the difference between the last two values of `average_BP`. This example fails in the pacing type analysis since 5 is not a GCD of 30. Since `average_BP` is evaluated every 30 minutes, there is no value every 5 minutes this output stream is accessed. The solution for this example is to switch the frequencies of the two output streams.

```
input blood_pressure : Int64
output average_BP @30min := blood_pressure.aggregate(over: 1min, using:
     average).defaults(to:0)
output avg_HR @5min := average_BP - average_BP.offset(by:-1).defaults(to:0)
```

△

Mixed Pacing Types In the user study, a common error was mixing event-based and periodic pacing types. The error hazard was usually synchronous lookups or offset lookups within periodic output streams to event-based streams. These lookups are valid from periodic to periodic streams that fulfill the frequency conditions discussed in the previous paragraph.

Example 4.6.6. This code snippet `alt_avg` computes the average altitude over one minute every second. The default expression is a synchronous lookup to the input stream `altitude`. Input streams always have an event-based pacing type according to the event-based timeline of the input stream. Thus, the correct default expression is `altitude.hold().defaults(to: 0)`.

```
input altitude: UInt64
output alt_avg : Bool @1s := altitude.aggregate(over: 1min, using:
     average).defaults(to: altitude)
```

△

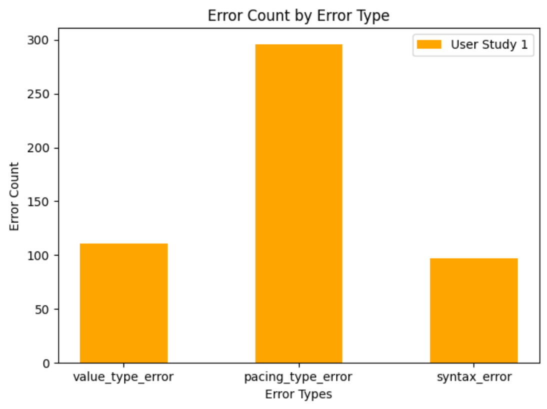


Figure 4.10: Results Error Types

4.6.3 Syntax Errors

The syntax errors range from simple spelling errors within the stream name or the keywords within an expression to more severe syntax errors where the type is misplaced, or some other keywords are either left out or misplaced.

Misspelled Keywords Several syntax errors occurred due to misspelled streams that were defined and did not fulfill the naming conditions. For example, the keyword `not` is not allowed at the beginning of a stream name. Further, timing units were given with the incorrect keyword, so instead of `@1s`, some participants wrote `@1sec`. Then, the participants made errors with the expressions. They either misspelled identifiers or keywords or left some keywords out. For example, instead of `hold()` they wrote `holds()` or instead of `defaults(to: 0)` they wrote `defaults(0)`.

Misplaced Keywords The participants made several errors where keywords or operators were left out or misplaced. For example, the participants forgot `:=` in the output stream declaration or used `:=` in a trigger declaration. Further, in the output stream declaration, keywords were defined incorrectly. The close clause requires only a when condition. In the user study, an error occurred where the close clause was introduced with a with the expression: `output prop3 spawn when too_low eval @(landing_zone && too_low)with landing_zone close with altitude > 40`. The correct syntax of the close cause is `close when altitude > 40`.

4.7 Conclusion

This user study identified common errors divided into three categories: value type errors, pacing type errors, and syntax errors. Figure 4.10 shows the distribution of →Figure 4.10, p. 31

the errors. The most prominent errors were pacing-type errors. These main cause for these errors were mixed pacing types where participants accessed event-based streams from periodic streams with a synchronous access. The second most common errors were value type errors. Mostly, these were caused by the omitted default value in expressions that required one. The user study also revealed that the participants had difficulties with the syntax of RTLola.

Chapter 5

Block-Based Syntax for RTLola

This chapter answers the second research question: *How can a block-based syntax be effectively adapted to RTLola?* We introduce SRTLola, the block-based syntax of RTLola, and present its design inspired by the results of the first user study. Further, we discuss the implementation of the block-based syntax and its integration into the RTLola playground. Lastly, we will introduce the translation from SRTLola to RTLola and how it is integrated into the monitoring process with RTLola.

5.1 Overview

This thesis extends the RTLola framework to support the creation of block-based specifications. As shown in Figure 5.1, the framework previously supported only text-based specifications. As highlighted in the framework pipeline, we have added a step for creating a block-based representation of the specification. After the user creates this block-based specification, it is translated into a text-based format. The framework then proceeds as before with the text-based specification.

➔Figure 5.1, p. 33

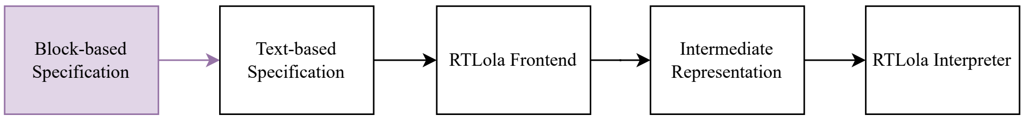


Figure 5.1: Overview of Framework with the Block-Based Syntax

5.2 Implementation of SRTLola

➔ Figure 5.2, p. 35

This section explains how we implemented the block-based syntax and how it is integrated into the RTLola playground. Figure 5.2 shows the playground with the block editor. The playground is split in the middle, with the block editor on the left and the text-based specification editor on the right. On the right, the user can switch between the specification, the trace, and the dependency graph. The block editor has a toolbox on the left where all categories that can be chosen are listed. The user can drag the blocks into the workspace and connect them. The workspace is a grid where the blocks can be connected to an SRTLola specification. The translate button in the workspace of the block editor is used to translate the block-based specification to a text-based one. The text-based RTLola specification is then displayed in the text-based editor. The user can then execute the specification and see the output in the console and the dependency graph.

Block Editor The block editor is implemented with Vue.js and is based on the Google Blockly library[1]. Each block in the toolbox is a custom block represented by a JSON object. The JSON object contains the type of the block, the category, the color, the shape, and the input fields of the block.

Limitations The block-based syntax is a prototype and does not support all features of RTLola. We cannot define parameters for output streams. We decided to focus on the RTLola fragment without parameters, as we faced a trade-off between the usability of the block-based syntax and the complexity of the language. Introducing parameters to SRTLola increases the complexity of the blocks. This would make the block-based syntax overwhelming for beginners, who are the target group of the tool.

5.3 Syntax of SRTLola

➔ Section 3.2, p. 10

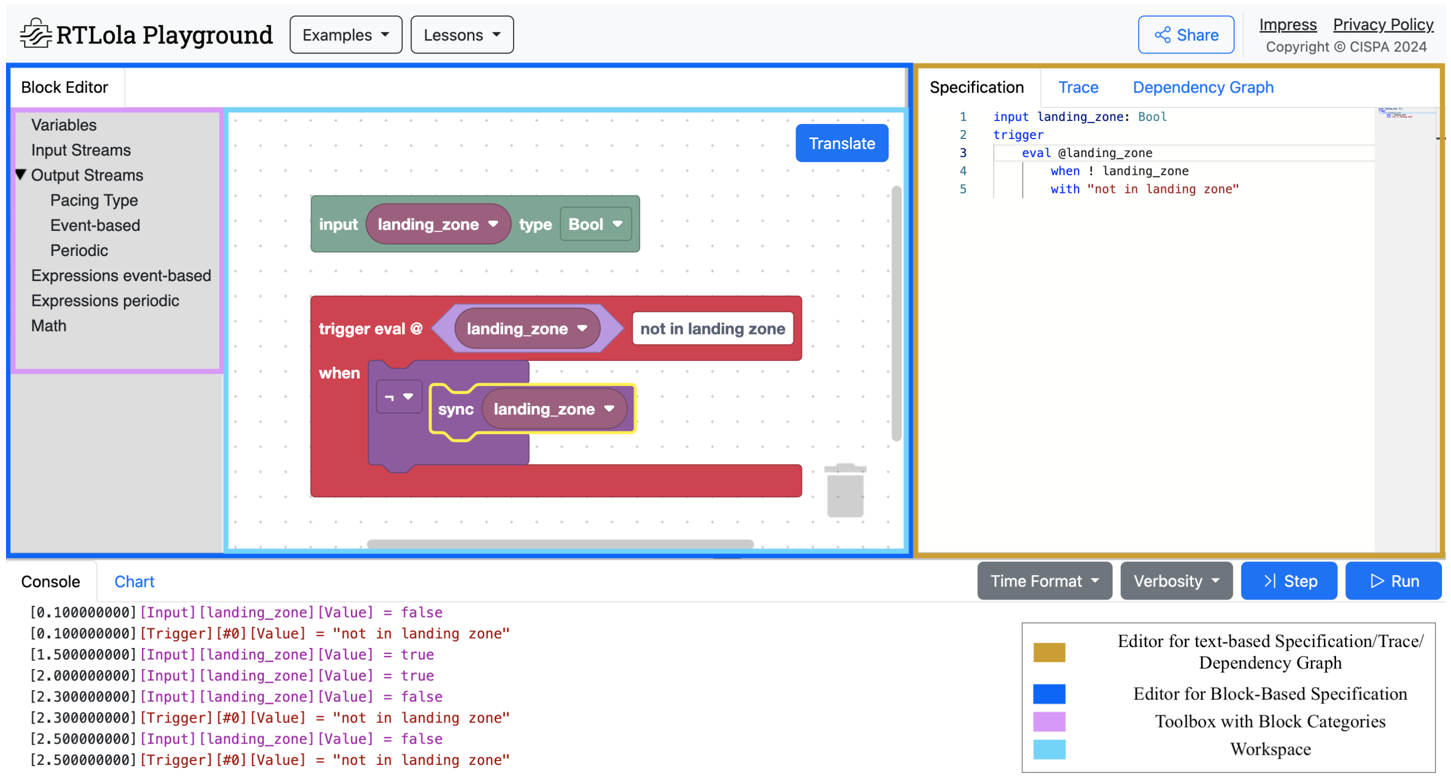
The syntax of SRTLola is inspired by the block-based syntax of Scratch introduced in Section 3.2. The name is inspired by Scratch, the most popular block-based language.

➔ Figure 5.2, p. 35

Block Categories The block-based syntax consists of different categories displayed in the block editor in Figure 5.2. Within the different categories, we have RTLola-specific blocks, such as input and output streams and expression categories. Output streams are divided into three subcategories: pacing type, event-based, and periodic. The expression categories are divided into periodic and event-based expressions. The Math category also includes arithmetic and boolean expressions.

[1]developers.google.com/blockly/

Figure 5.2: Playground with Block Editor



```
input landing_zone: Bool
trigger
  eval @landing_zone
    when ! landing_zone
    with "not in landing zone"
```

```
[0.100000000][Input][landing_zone][Value] = false
[0.100000000][Trigger][#0][Value] = "not in landing zone"
[1.500000000][Input][landing_zone][Value] = true
[2.000000000][Input][landing_zone][Value] = true
[2.300000000][Input][landing_zone][Value] = false
[2.300000000][Trigger][#0][Value] = "not in landing zone"
[2.500000000][Input][landing_zone][Value] = false
[2.500000000][Trigger][#0][Value] = "not in landing zone"
```

5.3.1 Design

The design of the SRTLola syntax uses different block types, shapes, and colors to represent the different components of the RTLola language. We start with an example to explain the block-based syntax intuitively. We will explain our design choices based on the errors presented in Chapter 4.

→Chapter 4, p. 15

Example 5.3.1. We consider the RTLola specification, which checks if a drone is in its landing zone. We release a trigger if the drone is not in the landing zone. Figure 5.3 shows the specification build with SRTLola.

→Figure 5.3, p. 37

```
input altitude: Int8
trigger eval @altitude when altitude.offset(by: -1).defaults(to: altitude) < 30
    "altitude below 30"
```

First, the specification consists of two streams: an input stream `altitude` and a trigger stream. These are represented by the two outer blocks in the SRTLola specification. The input stream contains an input block representing the stream name `altitude` and a dropdown menu for the value type, in this case `Int8`. The input block for the stream name `altitude` is used in the expressions to specify the referred stream of the expression. The trigger stream holds the pacing type with an input block with a lighter pink color. The input stream `altitude` is placed there to specify the pacing type. Lastly, expressions are statement blocks with a tree-based nature where the less-than operation is at the top level. On the next level, the offfset lookup and 30 are connected to the previous operator. Since the offset lookup needs a default value, the connection shape is a triangle. Thus, it can only be connected to the default block and not to the boolean operation. △

5.3.1.1 Stream Definitions

→Figure 5.4, p. 37

Every specification includes input streams, output, or trigger streams. Figure 5.4 shows that stream definitions are represented by outer blocks. The input streams are a special case of outer blocks since they do not contain a blank space for potential statement blocks. They only hold two input blocks for the stream name and the value type. In the variables category, the user creates, the user creates the input block of stream names. It is a rounded block and, after creation, is found in the variables category. The user can rename the stream globally. The output and trigger streams have additional statement inputs for the expressions and an input for the pacing type.

The outer blocks are independent of each other and cannot be connected. This captures the stream-based nature of the language by creating a new outer block for each stream of the specification. This is shown in Example 5.3.1, where both streams are represented by an outer block. Consequently, at the end of a block-based specification, we have multiple input, output, or trigger blocks thatforming the complete specification.

→Example 5.3.1, p. 36

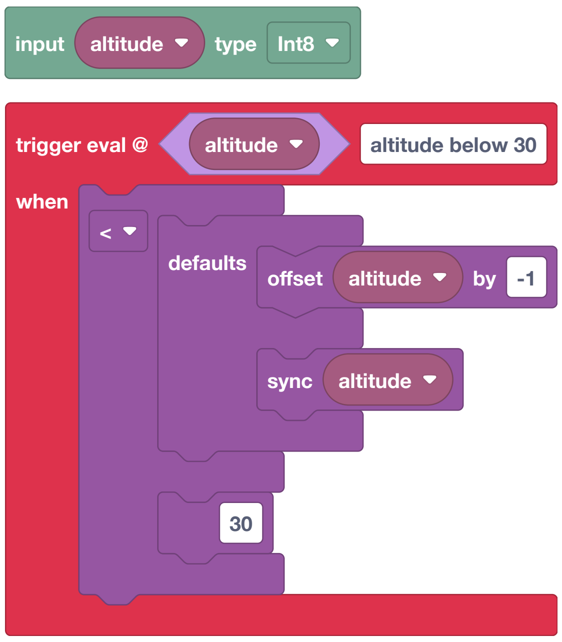


Figure 5.3: Example block-based specification

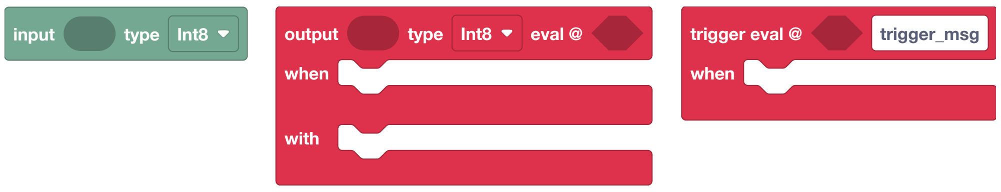


Figure 5.4: Different Stream Types

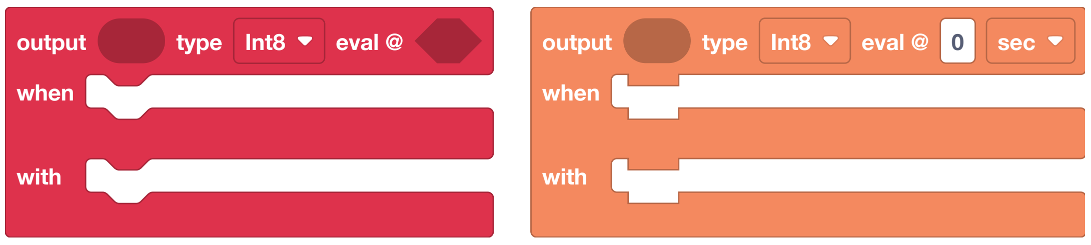


Figure 5.5: Event-based and Periodic Output Streams

5.3.1.2 Pacing Types

The pacing type of a stream is defined by the activation condition. As discussed in Section 4.6, the pacing type was the most common error type. Since pacing type errors are not of a syntactical nature, we elevate potential pacing type errors to the syntactical level. Therefore, all expression, output, and trigger stream blocks are divided into two categories, event-based and periodic, displayed in Figure 5.5. The stream definitions are split by their connection shape and color. Event-based output streams are red, and the connector shape is a half-diamond. Periodic output streams are orange, and the connector shape is rectangular. Respectively, the pacing type is represented by the same shape as the connector of the outer block. In event-based output streams the pacing type is represented by a diamond-shaped input block. In periodic output streams it is represented by a rectangular text input where the period is specified and a dropdown menu for time unit that needs to be selected.

➔ Section 4.6, p. 28
➔ Figure 5.5, p. 37

This split creates more awareness of the two major groups of pacing types. Expressions such as aggregations that are exclusively used in periodic output streams are therefore solely available in the periodic expression category. Further, each output stream block requires a pacing type. Consequently, pacing type errors in streams with solely hold lookups cannot occur. Moreover, this teaches the user to learn the concept of pacing types.

However, not every potential pacing type error can be captured by the syntax of SRTLola. The mixed pacing type errors and incorrectly annotated frequencies of periodic pacing types discussed in Section 4.6 will not be prevented by SRTLola. For example, if the user does a synchronous lookup from a periodic stream to an event-based stream, this error is not captured.

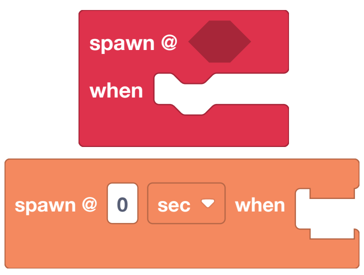
➔ Section 4.6, p. 28

5.3.1.3 Stream Lifecycle

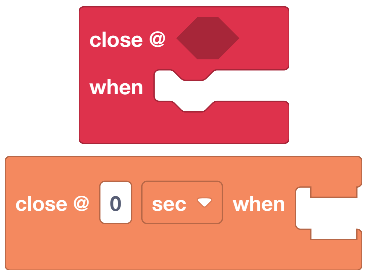
The stream lifecycle is represented by creating subclauses for spawning and closing a stream. Figure 5.6 shows that the spawn and close subclauses are divided by the event-based and periodic pacing type in addition to the stream definitions. Figure 5.7 shows how the spawn and close subclauses are attached to event-based trigger blocks. The left trigger contains just a spawn clause, whereas the right trigger stream contains both subclauses. The eval subclause is always at the top of the output or trigger block. Then, the spawn and close subclauses are attached to the eval subclause. Respectively, every possible combination of output and trigger streams with spawn and close subclauses is possible with SRTLola.

➔ Figure 5.6, p. 39
➔ Figure 5.7, p. 39

Remark 5.3.1. *The prototype only supports the combinations of purely event-based or periodic types for the spawn, eval, and close subclauses.*



(a) Spawn Subclause



(b) Close Subclause

Figure 5.6: Stream Lifecycle

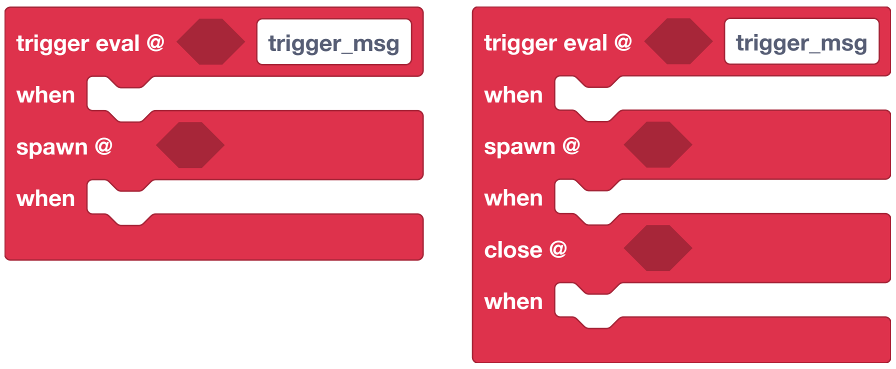


Figure 5.7: Trigger Blocks with Spawn and Close Subclauses

5.3.1.4 Semantic Types

The semantic types of RTLola are represented by a blank space where an expression can be dragged into. Figure 5.5 illustrates that the semantic type is indicated by the keyword *when*. The user can then specify the semantic type by connecting the statement blocks to the *when* blank space. Semantic types are accessible in all subclauses of the stream lifecycle.

→Figure 5.5, p. 37

5.3.1.5 Expressions

Along with the stream definitions, expressions are divided into two categories: event-based and periodic expressions. All expression blocks are represented by statement blocks. Figure 5.8 shows all RTLola expressions with the respective SRTLola blocks.

→Figure 5.8, p. 40

Section 4.6 revealed that the participants struggled with the default expression. A common error was the omitted default expression in the category of value type errors. We introduce a different connection shape for expressions that need a default value to unfold where the default expression is necessary. Expressions with a default value, such as the offset lookup and hold lookup, have a triangular connection shape. A special

→Section 4.6, p. 28

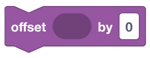
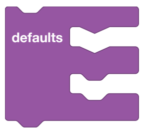
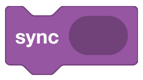
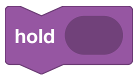
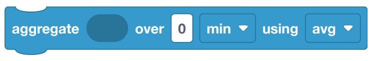
Name	RTLola Syntax	Block-Based Syntax
Offset	`sref.offset(by: -n)`	
Default	`e.defaults(to: e)`	
Synchronous Lookup	`sref`	
Sample&Hold Lookup	`sref.hold()`	
Sliding Window	`sref.aggregate(over: δs, using: γ)`	

Figure 5.8: RTLola Syntax and SRTLola Blocks

case is the aggregation expression, designed only in the periodic blocks. We have two types of aggregation blocks, one with a default value and one without.

5.4 Translation of SRTLola to RTLola

The translation from SRTLola to RTLola is a direct translation. It begins by gathering all blocks from the workspace and processing them. Each group of input, output, or trigger streams is translated into the corresponding RTLola syntax. Each block contains a translation function that processes the block and all connected blocks such as input and statement blocks, and converts them to the RTLola syntax. Figure 5.8 illustrates the mapping of RTLola expressions to SRTLola blocks used during the translation of expressions. The list of RTLola stream definitions alongside the corresponding SRTLola blocks is presented in Figure 5.9. Note that only event-based blocks are shown. Analogously, the translation of periodic blocks follows the same process, except for converting the pacing type to a periodic type.

→Figure 5.8, p. 40
→Figure 5.9, p. 41

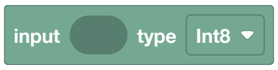
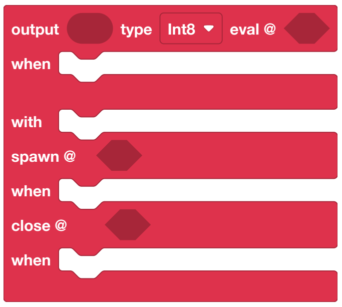
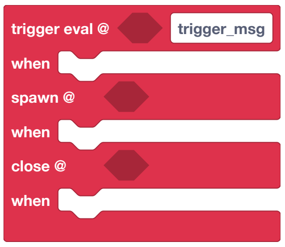
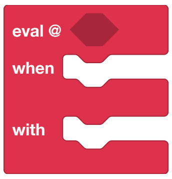
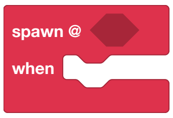
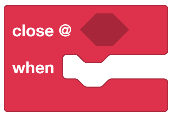
	RTLola Syntax	Block-Based Syntax
Input	`input sname : ValueType`	
Output	`output sname : ValueType spawnclause evalclause closeclause`	 output type Int8 eval @ when with spawn @ when close @ when
Trigger	`trigger spawnclause evalclause closeclause "trigger_msg"`	 trigger eval @ trigger_msg when spawn @ when close @ when
Eval Subclause	`eval @activation_cond when cond with expr`	
Spawn Subclause	`spawn @activation_cond when cond`	
Close Subclause	`close @activation_cond when cond`	

Figure 5.9: RTLola Translation of SRTLola Blocks

Chapter 6

Evaluation

This chapter answers the third research question: *Does the block-based syntax approach help to reduce common errors in RTLola specifications?* We evaluate the block-based syntax by conducting a second user study. The study design is similar to the first user study. The goal of the user study is to test the prototype of SRTLola and evaluate the usability of the block-based syntax. We analyze the errors made by the participants and compare them to those made in the first user study.

6.1 Study Design

This user study aims to test the prototype of SRTLola and evaluate the usability and performance of the block-based syntax. Further, we evaluate which errors do not occur with the block-based syntax we found during the first user study (see Chapter 4). →Chapter 4, p. 15

We conducted the user study with six participants (four male, two female, and an average age of 25). As in the first user study, all participants have a background in computer science, either as undergraduates or as PhD students in computer science.

Remark 6.1.1. *Since RTLola is a complex language, learning it takes time. Therefore, we set a time limit of 90 minutes for the user study to avoid exhaustion, as in the first user study. Thus, not every participant could finish all tasks, but every participant finished at least the first three tasks.*

6.1.1 Procedure

As in the user study described in Chapter 4, the participants conducted the user study in a room with the researcher. →Chapter 4, p. 15

6.2 Tasks

The participants had to solve the same tasks as in the previous user study. The tasks are depicted in Section 4.5. The difference is that the participants had to solve the tasks with the block-based syntax.

➔Section 4.5, p. 18

6.3 Results and Discussion

During the user study, the participants faced some challenges that resulted in errors. All errors were related to the value or pacing type analysis of RTLola. Only two errors were on the syntax level. However, one error only occurred due to a bug in the prototype mentioned in Remark 6.3.1. We discuss the errors in the following sections similar to Section 4.6. We divide the errors into three categories: value type errors, pacing type errors, and syntax errors.

➔Remark 6.3.1, p. 45
➔Section 4.6, p. 28

6.3.1 Value Type Errors

In the block-based syntax of RTLola, we have to specify the value type for input and output streams by selecting the type in the dropdown menu. Figure 5.4 shows that the default type of an input and output stream block is always `Int8`.

➔Figure 5.4, p. 37

Example 6.3.1. The code snippet shows that the output stream compares the last value of altitude and checks if it is greater than 30. Since this evaluates either to true or false, the type of the output stream should be `Bool`.

```
output difference: Int8
   eval when too_low with - altitude.offset(by: -1).defaults(to: altitude) > 30
```

△

6.3.2 Pacing Type Errors

The participants had problems understanding the concept of pacing types, especially those who had never worked with RTLola before. This resulted in different errors. The most common error was due to mixing up periodic and event-based streams. The following two examples show pacing type errors due to an incorrect given pacing type.

➔Figure 6.1, p. 45

Example 6.3.2. In Figure 6.1, the participant wanted to specify the pacing type of the trigger stream. Since in RTLola, event-based pacing types are only defined over input streams, the correct pacing type for the trigger stream has to be defined with the input stream `landing_zone`. This resulted in a pacing type error since `too_low` is an output stream. △

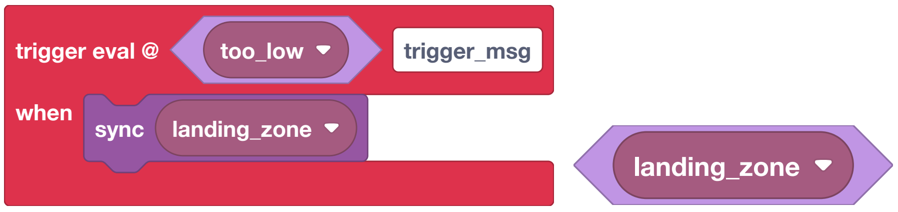


Figure 6.1: Wrong pacing type

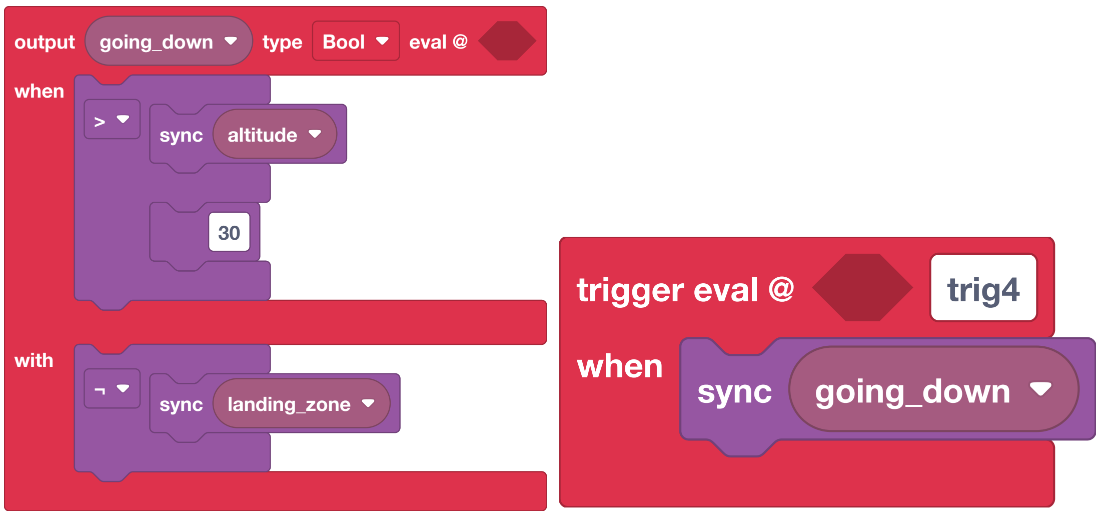


Figure 6.2: Semantic Pacing Type Error

As Example 6.3.2 shows, SRTLola is not able to guide the user towards using explicitly input streams for event-based pacing types. This was a design choice when implementing the prototype. Usually, the event-based pacing type can be omitted since it can be inferred. However, if the output stream only contains hold lookups, the pacing type has to be given explicitly. Once

→Example 6.3.2, p. 44

Example 6.3.3. In Figure 6.2 the participant specified trigger `trig4` with a synchronous lookup to the output stream `going_down`. The stream definition of that output stream states with the statement block in the when condition that the stream has a semantic type of `altitude > 30`. This concludes that the trigger accessing the output stream must satisfy the semantic type. Therefore the trigger `trig4` has to be extended with the condtion `altitude > 30`. The correct SRTLola block is shown in Figure 6.3. △

→Figure 6.2, p. 45

→Figure 6.3, p. 46

6.3.3 Syntax Errors

Remark 6.3.1. *We can connect periodic blocks after one another. This is a bug in the prototype and should not be possible.*

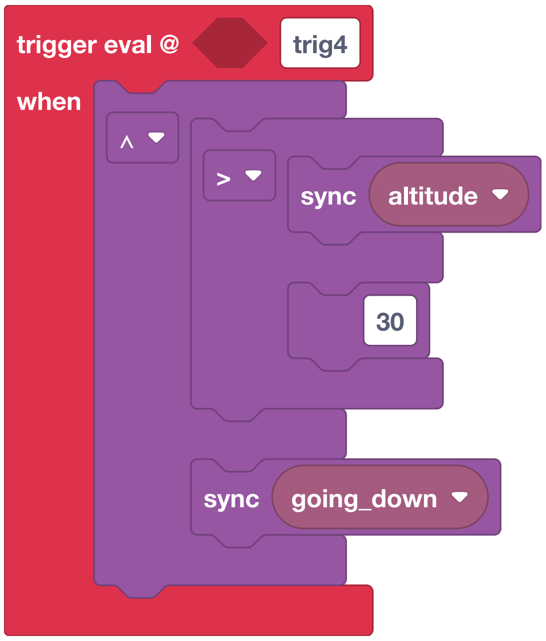


Figure 6.3: Trigger with correct semantic type

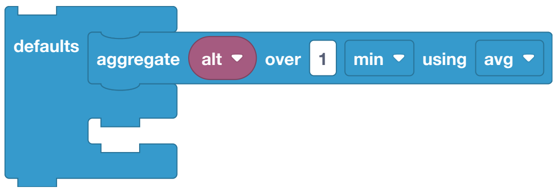


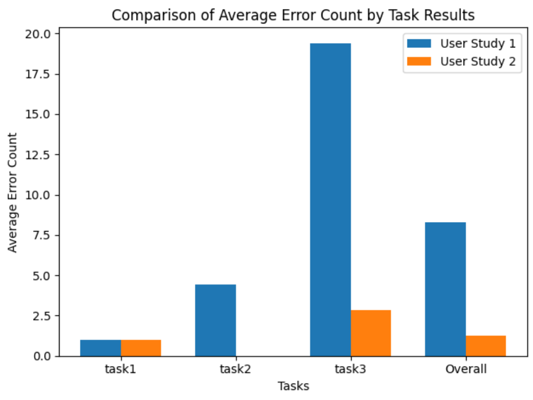
Figure 6.4: Missing block in the default block

→Remark 6.3.1, p. 45
→Figure 6.4, p. 46

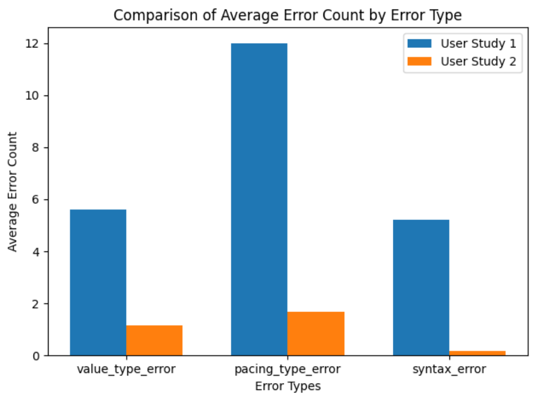
There were only two syntax errors throughout the user study. The first syntax error was due to the bug mentioned in Remark 6.3.1. The second error was a missing block in the default block. This error occurred in task 4. In Figure 6.4, the block snippet that causes the error is illustrated. While doing the aggregation, the block for the default value is missing.

6.3.4 Conclusion

The second user study showed that the block-based syntax is helpful for beginners learning RTLola. Compared to the first user study, the participants made fewer errors. The errors that occurred were mainly due to the value and pacing type analysis. The participants appreciated the different connection shapes and colors of the blocks, which guided them in the building process.



(a) Average Error Count by Task



(b) Average Error Count by Error Type

Figure 6.5: Quantitative Analysis

6.3.4.1 Quanitative Analysis

To compare the results from both user studies, we analyze the average error count by task and the average error count by error type. Since the sample sizes of the two studies vary and not all participants in the second study completed every task, we compare only the first three tasks and calculate the averages. Figure 6.5a shows the average error count by task. While the averages for task one are identical in both studies, the average error count for task two in the second study is zero. This indicates that the block-based syntax eliminated all errors observed in the first study. The reduction in the average error count is even more pronounced. ➔Figure 6.5a, p. 47

Figure 6.5b illustrates the average error count categorized by error type. Compared to the initial user study, the average error count for all error types is lower. The most common error type in both studies is pacing type errors. However, the user study using SRTLola shows a significant reduction in pacing type errors, bringing the average count close to that of value type errors. Although we do not provide a solution for all pacing type errors, the distinction between event-based and periodic blocks helps to reduce these errors. Additionally, SRTLola eliminated all omitted default errors observed in the first user study, resulting in a significant decrease in value type errors. Furthermore, almost all syntax errors were resolved in the second user study; the only syntax error occurred due to a missing block in the default block. ➔Figure 6.5b, p. 47

Overall, SRTLola proves to be a valuable tool for learning RTLola. Its block-based syntax simplifies the learning process and minimizes the number of errors made by users.

Chapter 7

Conclusion

This thesis introduced the block-based syntax SRTLola to lower the entry barrier for new users learning the RTLola specification language. We addressed three research questions that reflect our three main contributions.

First, the first user study of this thesis revealed common errors while writing RTLola specifications. The most prominent errors were pacing type errors.

Second, we designed the block-based syntax SRTLola. Our design choices were driven by the results of the first user study, focusing on the common errors. We divided all blocks into either event-based or periodic blocks. This is the first step for users to understand which expressions can only be used in the periodic or event-based contexts. We implemented a prototype of SRTLola and integrated it into the RTLola playground, enabling the writing of specifications in block-based syntax that can be translated into text-based specifications. However, the prototype does not yet support all RTLola features, such as parameters.

Lastly, a second user study we conducted found that the block-based syntax approach reduces common errors while learning RTLola. The trend in the quantitative analysis indicates that participants made fewer errors overall with SRTLola compared to the text-based syntax of RTLola. Participants provided positive feedback, finding the block-based syntax intuitive and easy to use.

7.1 Future Work

For future work, there are multiple interesting approaches to how the tool can be improved and extended.

Extended User Studies In this thesis, we conducted user studies with a small number of students and focused on the qualitative feedback of the studies. We could conduct a more extensive user study with more participants for future work to get more statistically significant results. Further, in this thesis, we analyzed both approaches independently. We could conduct a user study comparing both approaches, the text-based and the block-based one. The comparing metrics could be the time needed to write the specification, the number of errors, and the number of successfully completed specifications. Since SRTLola aims to teach the RTLola language with the prospect of writing text-based specifications, we could conduct a long-term study to see if the block-based syntax helps to learn the text-based language faster than just the text-based tutorial.

Real-time translation of SRTLola Instead of translating the block-based specification after it is built, it could be done in real-time. The text-based specification is shown every time a new block is dragged into the workspace. This would allow the user to see the corresponding text-based representation while building the block-based specification. Further, this would improve the users' understanding of the connection between block-based and text-based representation.

Displaying Errors So far, the prototype shows the error messages of the RTLola frontend in the console based on the text-based specification. We could adapt the prototype to match the error message to the corresponding block that causes the error and highlight the block by changing the color or shape. This would aid the user to find the block that is causing the error. Moreover, the block-based syntax approach could be extended to other specification languages, particularly stream-based languages like TeSSla. This expansion could open up new avenues for the application of SRTLola. However, it's important to balance the complexity of the language with the simplicity of the block-based syntax. Introducing all features at once could overwhelm users, especially beginners. Our ultimate goal is to teach the language so that users can learn to write text-based specifications.

Validating the Block-based Syntax We could further validate each block placement action in SRTLola. We would need to introduce a validation algorithm that checks if the block would cause an error. This algorithm cannot rely entirely on the evaluation of the translated text-based specification. If the user starts to build the specification and adds a stream name to an output block, the expression of that block will still be missing. The evaluation of the text-based specification would result in a syntax error, but as the user proceeds, the output block could become valid. Therefore, the validation algorithm should have an underlying symbolic execution to check if the specification could eventually become valid. The action should be rejected only if there is no way of creating a valid specification with the action.

Bibliography

[AK17] Ernest Afari and Myint Swe Khine. "Robotics as an educational tool: Impact of lego mindstorms". In: *International Journal of Information and Education Technology* 7.6 (2017), pp. 437–442.

[Bau+25] Jan Baumeister et al. "A Tutorial on Stream-Based Monitoring". In: *Formal Methods*. Ed. by Andre Platzer et al. Cham: Springer Nature Switzerland, 2025, pp. 624–648. ISBN: 978-3-031-71177-0.

[Buc+24] Alessio Bucaioni et al. "Programming with ChatGPT: How far can we go?" In: *Machine Learning with Applications* 15 (2024), p. 100526. ISSN: 2666-8270. DOI: `https://doi.org/10.1016/j.mlwa.2024.100526`. URL: `https://www.sciencedirect.com/science/article/pii/S2666827024000021`.

[DFS21] Johann C. Dauer, Bernd Finkbeiner, and Sebastian Schirmer. "Monitoring with Verified Guarantees". In: *Runtime Verification*. Ed. by Lu Feng and Dana Fisman. Cham: Springer International Publishing, 2021, pp. 62–80. ISBN: 978-3-030-88494-9.

[Doc20] Simulink Documentation. *Simulation and Model-Based Design*. 2020. URL: `https://www.mathworks.com/products/simulink.html`.

[Dur+07] M.J. Duran et al. "A learning methodology using Matlab/Simulink for undergraduate electrical engineering courses attending to learner satisfaction outcomes". In: *International Journal of Technology and Design Education* 17 (Mar. 2007), pp. 55–73. DOI: `10.1007/s10798-006-9007-z`.

[FKS23] Bernd Finkbeiner, Florian Kohn, and Malte Schledjewski. "Leveraging Static Analysis: An IDE for RTLola". In: *International Symposium on Automated Technology for Verification and Analysis*. Springer. 2023, pp. 251–262.

[FPR12] Teresa Ferrer-Mico, Miquel Àngel Prats-Fernàndez, and Albert Redo-Sanchez. "Impact of Scratch Programming on Students' Understanding of Their Own Learning Process". In: *Procedia - Social and Behavioral Sciences* 46 (2012). 4th WORLD CONFERENCE ON EDUCATIONAL SCI-

ENCES (WCES-2012) 02-05 February 2012 Barcelona, Spain, pp. 1219–1223. ISSN: 1877-0428. DOI: https://doi.org/10.1016/j.sbspro.2012.05.278. URL: https://www.sciencedirect.com/science/article/pii/S1877042812014073.

[Gro20] Iwona Grobelna. "Scratch-Based User-Friendly Requirements Definition for Formal Verification of Control Systems". In: *Informatics in Education* 19 (June 2020), pp. 223–238. DOI: 10.15388/infedu.2020.11.

[Kal+22] Hannes Kallwies et al. "TeSSLa – An Ecosystem for Runtime Verification". In: *22nd International Conference on Runtime Verification (RV)*. Springer International Publishing. Tbilisi, Georgia: Springer International Publishing, Sept. 2022. DOI: 10.1007/978-3-031-17196-3_20. URL: https://link.springer.com/chapter/10.1007/978-3-031-17196-3_20.

[Mal+10] John Maloney et al. "The Scratch Programming Language and Environment". In: *ACM Trans. Comput. Educ.* 10.4 (Nov. 2010). DOI: 10.1145/1868358.1868363. URL: https://doi.org/10.1145/1868358.1868363.

[Pnu77] Amir Pnueli. "The temporal logic of programs". In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[WW15] David Weintrop and Uri Wilensky. "To block or not to block, that is the question: students' perceptions of blocks-based programming". In: *Proceedings of the 14th International Conference on Interaction Design and Children*. IDC '15. Boston, Massachusetts: Association for Computing Machinery, 2015, pp. 199–208. DOI: 10.1145/2771839.2771860. URL: https://doi.org/10.1145/2771839.2771860.