Modularization of code in RTLola

Saarland University
Department of Computer Science
Bachlor's Thesis

submitted by
Jan-Robin Aumann

Saarbrücken, October 2022

Supervisor:	Professor Bernd Finkbeiner, Ph.D.
Advisors:	Florian Kohn, B.Sc.
	Jan Baumeister, M.Sc.
Reviewer:	Professor Bernd Finkbeiner, Ph.D.
	Professor Martina Maggio, Ph.D.
Submission:	October 26th, 2022

Abstract

Large codebases and complex software projects are often hard to maintain. One of the factors making them this way is code organization.
Poor organization of code leads to more duplication when engineers can't easily find existing implementations of functionality.
Depending on the programming language and toolchain used, they might offer different ways of combatting this problem by offering ways to structure the code according to the principles of modularity.
This work solves the problem for specification languages such as RTLola [11] by introducing a concept of modularization suited to stream-based languages and an implementation of that conecept for RTLola. We propose an implementation of modules that enables reusable and exchangeable code through modules, interfaces and instances.

Foreword

I would like to express my sincere gratitude to the people that got me to this point. Starting with Professor Bernd Finkbeiner for offering me this interesting thesis topic; my advisors Florian Kohn and Jan Baumeister for their guidance, support, and patience in assisting me on this thesis, and providing invaluable feedback.
Furthermore, I would like to thank Professor Bernd Finkbeiner and Professor Martina Maggio for reviewing this thesis.
Lastly, I would like to thank to my parents for supporting me in studying a subject I am passionate about, my fiancée for helping me to stay optimistic each and every day and my friends for their support. I particularly would like to thank my father-in-law, Ken O'Keefe, as well as Kim Schneider, Winfried Kramer and again my fiancée Shannon O'Keefe for proofreading this thesis.

Statement in Lieu of Oath/Eidesstattliche Erklärung

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

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

Declaration of Consent/Einverständniserklärung

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

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

Jan-Robin Aumann
Saarbrücken, den 26.10.2022

Contents

Chapter 1

Introduction

Large scale software projects, such as the Linux kernel - made up of 22583597 lines of code split out over 63522 individual files - contain a lot of complexity. Left unstructured, that complexity eventually grows into an unmanageable, unmaintainable mess of code, that is hard to reuse and keep free of duplication. Such code bases are not only hard to maintain, but they are also difficult to survey and understand for new developers joining the team.
Modularization of code is an essential tool in achieving a successful, well structured result when tackling a software project of all but the smallest sizes. It is a solution supported by almost all modern programming languages that helps to keep rising complexity at bay. It allows for compartmentalization of individual logical units of source code into modules in which the complexity can be kept low. This yields - relative to the project size - small and simple modules that are both easier to understand and maintain for individual software developers than the whole project.
Modularization affords another benefit: Interchangeability. One module with an implementation of an interface can be easily and effortlessly exchanged with another module fulfilling the requirements of the same interface.
This same problem arises when defining specifications for monitoring real-time properties in stream-based languages such as RTLola [11]. RTLola is a memory and execution time bounded specification language that can express complex real-time properties by interpreting input data as streams to be fed into a monitor that computes output streams. For example, one might want to structure a specification monitoring the operation of a UAV into components that monitor specific subsystems of the aircraft and reuse them instead of duplicating the parts of the specification.
Currently in RTLola it is not possible to reuse code in any way. Inputs, outputs and triggers have to be declared upon usage and if multiple streams with a similar or equal functionality are needed, they have to be declared separately. In this thesis we solve this by introducing a concept of modularization to RTLola. We also take a look at the state of the art of modularization in other synchronous programming languages, discuss our approach and implementation of modularization and evaluate our results with a case study on actual users of RTLola.
We propose an implementation that would allow us to reuse and exchange code at will by instantiating modules that implement an interface specification such

that different modules implementing the same interface can be exchanged.
We also see the opportunity for optimization. If a module is imported and or instantiated twice we would be able to reuse the artifacts from the first occurrence instead of reinterpreting the imported module.

Chapter 2

Related work

As RTLola [11] is a stream based specification language it is different in some regards to a traditional programming language, which also means that not all concepts of structuring and modularization transfer 100% to it and may need to be adapted.
Most programming languages follow the imperative programming style, where the program is a sequence of instructions [20].
By contrast, RTLola is closer to the event-driven programming style, where user input determines the programs flow. But in the case of stream based specification languages, the arrival of data points on the stream, which can be either periodic or event-based, determines this.
A great deal of work has been done in an effort to reduce complexity in software. First efforts were undertaken by Böhm and Jacopini [3], by postulating that any computable function can be expressed by combining subprograms either in sequence, iteration or selection. This laid the groundwork for others like [8] and [6] to found the field of structured programming, a paradigm that forgoes any jumping around to labels (with the help of the `goto` statement) in favor of structured control flow for selection, repetition as well as the execution of subroutines. This is done in an effort to improve clarity, quality and development time of programs. Clearly, this does not apply directly to our work but is an important stepping stone in the evolution of programming languages.
This theory is still commonly in use today with one notable exception; most modern programming languages provide ways to abort both the execution of a subroutine and a loop execution early with statements like `return`, `break` and `continue` [21]. RTLola does not have this, since it doesn't operate on functions with a point of entry/(multiple) point(s) of exit, but instead operates continuously on monitored streams.
As a larger scale analogue one can see modular programming as it arose from [1]. Here, modular programming is referred to as a method of code reuse via software libraries. This is highly interesting to us since many of the concepts directly transfer to our approach.
Later other important concepts were included like Information Hiding [18], a principal technique to hide the parts of a program that are likely to change and exposing those that are not, thus promoting stability. Another important addition is Separation of Concerns [9], a design principle for software where programs are separated into distinct sections, each of which only deals with a

certain concern, e.g. persistence layer, data access layer and business logic layer. Some languages have been built from the ground up with the thought of modular programming in mind [23], like Modula-2, and most modern languages, like Go, C#, JavaScript, etc. have ways to separate programs into modules according to the theories of the previously mentioned works [16], [12], [10]. The most complete module system is StandardML [17], incorporating all the principles mentioned previously and even parameterized modules, also known as functors. A relatively young language, Rust [22], has a simple but powerful module system to organize code hierarchically. Here, a module is a collection of items like functions, structures and even other modules. However, the implementation details are not required to be split from the descriptive items.
The way RTLola describes systems via streams is very similar to how Hardware Definition Languages like Verilog and VHDL describe systems at the level of circuits via signals. They have different terms for signals (`wire`s and `register`s respectively) and Verilog additionally provides a way to express signal strengths, but functionally they both allow representing signal values in a many-valued logic. Similarly, RTLola allows the representation of real-time properties over streams as stream expressions.
Verilog [13] has modules, which optionally declare a port map - akin to input/output streams - and functionality within their scope. These can then be instantiated inside other modules to enable reuse of the functionality they provide. They can also be wired together such that two modules may communicate and work with each other's results.
VHDL [14] on the other hand has two different constructs that work together to achieve a similar result: A unit of functionality is first described in an entity without providing implementation details. Only afterwards is an architecture provided which fills that description with an implementation.
Both of these are a great inspiration to us, since they enable reuse of previously declared functionality, promote separation of concerns and - in the case of VHDL - segregate description and implementation. Both Esterel [4] and Lustre [5] offer the ability to specify subprograms. Once declared they can be instantiated. Neither of these concepts of modularity allows splitting the codebase into different files, nor do they allow for any type of interface that a implementation has to fulfill. Other specification languages like Alloy [15] have also adopted these theories and implemented a module system, proving its usefulness in specification languages.

Chapter 3

Background

In this background chapter, we will discuss the RTLola language and its predecessor Lola with a focus on how they - especially the former - is typed, evaluated and analyzed throughout its compiler frontend, as laid out in both the original paper about Lola [7] and the follow-up paper about RTLola [11] as well as the source code of the RTLola frontend.

3.1 RTLola

RTLola [11] is a stream-based specification language to monitor cyber-physical systems in real time. It is an extension of Lola [7], a specification language created to monitor synchronous systems.
Specifications in RTLola consist of a series of definitions of input streams, output streams and triggers. These specifications are evaluated to generate runtime monitors, which can then be used to monitor the system it was modeled on.

3.1.1 Streams

A stream is a finite sequence of data points with a uniform type. Types will be explained in greater detail in the later Section 3.1.7 on the type system.
Input streams consist of values to be monitored, e.g. the temperature of a CPU in a computer. They are descriptive of the current state of the monitored system.
Output streams compute their data points by evaluating stream-expressions defined in the specification over current and previous data points from other streams. We will discuss the different ways to access other streams in Section 3.1.3.
RTLola makes no assumption on the arrival frequency of data points in the input streams. This is in contrast to its predecessor Lola, which imposed the restriction that all streams had to be synchronized by a global clock. While this is perfectly acceptable when monitoring synchronous circuits, it is not when monitoring real-time systems with sensors that might produce readings at different frequencies or even just when specific events occur.
The following example specification monitors measurements of a temperature probe of a CPU. These values are then used to make a Boolean statement whether or not the temperature is over 60 degrees.

```
input cpu_temp : Float32
output temp_over_60 := cpu_temp > 60.0
```

Listing 3.1: Specification to monitor the temperature of a CPU

The monitored values are accessible through the input stream `cpu_temp`. This stream is accessed in the stream-expression of the output stream `temp_over_60`, which takes values from the input stream, compares them against a constant of `60.0` to determine if the current temperature is greater than 60 degrees.
This will produce an output of `false` as long as the CPU temperature remains under 60 degrees. If it rises over that limit, the output will read `true`.

3.1.2 Triggers

Triggers are a way of detecting and alerting when a specification has been violated. A trigger is defined by a Boolean stream-expression to determine if the specification has been violated and a message to describe the violation. Once the expression evaluates to `true`, the trigger is executed.
We extend our previous example of the CPU temperature monitor to alert us when the temperature is over the limit of 60 degrees.

```
input cpu_temp : Float32
output temp_over_60 := cpu_temp > 60.0
trigger temp_over_60 "The cpu temperature is over 60 degrees!"
```

Listing 3.2: Specification to monitor the temperature of a CPU with a trigger

We can announce breaches of the specification with a trigger that alerts once the temperature is in excess of 60 degrees with the message 'The CPU temperature is over 60 degrees!'. We can just use the output stream as the Boolean expression, since it already produces Boolean values that states whether or not the temperature is over the limit.

3.1.3 Stream access

In a stream-expression we can access other streams, meaning we can use the current, past or aggregated value(s) of any stream in our computation of another stream. This access can either be performed synchronously - meaning the timing of the two streams is bound together - or asynchronously, which leaves their timing uncoupled.

Synchronous access

Synchronous access binds the timing of streams, meaning that the accessing stream calculates a new value exactly when (all of) the accessed stream(s) have a new value available. In the previous examples 3.1 and 3.2 we synchronously accessed the last value of the `cpu_temp` stream. We can also access previous values and - if a value doesn't exist - provide a default value.
In the following example, we monitor RAM usage inside our system. Our available RAM is kept track of and whenever a program launches, the amount of memory it reserves is subtracted from the total. When a program terminates, it

again reports the amount of RAM freed, but negates that amount so it is added back to the total available amount.

```
input reserve_or_free : Int64
output available :=
    available.offset(by: -1).defaults(to: 16384) - reserve_or_free
trigger available < 0 "No more RAM available to reserve"
```

Listing 3.3: Specification to track the reserved memory in a system

The input stream `reserve_or_free` gets a new data point when a launched program provides the amount of RAM it needs to reserve (positive value) or if a program shuts down and frees the amount (negative value) used by it.
We keep track of the available RAM with the `available` output stream, which calculates a new value every time that `reserve_or_free` receives a value. The new value for `available` is the previous value minus the amount that was reserved/freed.
The initialization with the amount of system RAM, in this example 16 Gigabyte, is achieved by having the `defaults` operator set to the amount of RAM. This way, the first time a program registers and the previous value for available does not exist, it uses the default amount.

Asynchronous access

Asynchronous access by contrast decouples the timing of individual streams. That means the arrival of a new value in an accessed stream does not cause the computation of a new value in the accessing stream.
For an example of asynchronous access refer to 3.4

3.1.4 Periodic and event-based streams

All the streams we have seen so far were event-based, meaning they calculated new values when values were received in (all) accessed streams (more on this in Section 3.1.5 on activation conditions). RTLola also has periodic streams, which are evaluated at a specified frequency.
In the following example, we extend example 3.3 to output the percentage of available RAM once every second.

```
input reserve_or_free : Float32
output available := available.offset(by: -1).defaults(to: 16384.0) -
    reserve_or_free
trigger available < 0.0 "No more RAM available to reserve"
output percentage_used @1Hz := (available.hold().defaults(to:
    16384.0) / 16384.0) * 100.0
```

Listing 3.4: Specification to periodically output the percentage of available memory

To output the percentage of available RAM we first have to change the data type of the streams from Integer to Float (more specifically from `Int64` to `Float32`) so that we can divide the available amount by the total amount and have the result be a floating point number and not just 1 or 0. More on types will be

discussed in Section 3.1.7.
Now we can introduce a new (periodic) output named `percentage_used` and annotate it with our desired evaluation frequency of 1 Hertz. In its stream-expression, we asynchronously access the `available` stream via the `hold` operator and do our computation of the available percentage by dividing by the total amount and multiplying the result by 100.

3.1.5 Activation conditions

An activation condition is the defining expression that states when the monitor calculates a new value for a given stream. It is a positive Boolean expression over stream names that can be either conjuncted or disjuncted. For the former, the monitor only calculates a new value for the stream if all the streams in the activation condition receive a new value. Similarly for a disjunctive activation condition, if any of the streams in it receive a new value, the activation condition evaluates to true.
If no activation condition is explicitly annotated, it is inferred based on the accessed streams in the stream expression.

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

Listing 3.5: Example of inferred activation condition on streams

In this example, the activation condition for the output stream `c` is inferred as $a \wedge b$, while the activation condition for the output stream `b` is trivially just a. We can of course also use just a stream name as the simplest version of an activation condition. In the following example we want to keep track of how many values were received:

```
input in : Int64
output count @in := count.offset(by: -1).defaults(to: 0) + 1
```

Listing 3.6: Example of annotated activation condition with a stream name

Here we explicitly annotate a stream name (`@in`) as the activation condition for `count`. This makes the monitor add 1 to `count` every time a new value arrives for `in`. If the activation condition were not explicitly annotated, `count` would just remain inactive since the inferred activation condition would just include `count` itself.

3.1.6 Sliding windows

Additionally, RTLola is capable of computing sliding windows. The following example illustrates that by computing the average temperature of the CPU over the last minute.

```
input cpu_temp: Float32
output average @ 1Hz := cpu_temp.aggregate(over: 60s, using:
    average).defaults(to: 0.0)
```

Listing 3.7: Example of a sliding window over the last minute

With the `aggregate()` operator we can create a sliding window over a given timeframe specified with the `over` parameter using predefined aggregate functions like `average`, `sum`, `count`, etc. given to the `using` parameter.

3.1.7 Types

There are two different kinds of types that apply to each stream that dictate which information it carries and when that information will be available.
The former is called the value type and encodes the type of values conveyed by the stream, like `Int64`, `Float32` or `Bool`. Within a stream expression we can only use values of compatible types.
The latter is called the pacing type and encodes when the monitor will calculate a new value for the stream. In the case of event-based output streams the pacing type is the activation condition 3.1.5. In the case of periodic streams, the pacing type is the period that is derived from the frequency of the stream.

3.1.8 Evaluation

RTLola uses a dependency graph to determine the evaluation order of streams in the monitor.
This dependency graph makes use of a weighted and directed graph where vertices represent the streams of the specification. Edges from one vertex to another represent a stream access from the first to the second, where the weight of the edge is equal to the offset of the access.
Therefore we can say that a vertex representing an input stream will have no outgoing edges while triggers will have no incoming edges, since the former will never depend on any stream while the latter can't be accessed by other streams.
We examine a subset of RTLola excluding sliding windows and periodic streams for the sake of simplicity. RTLola's complete definition can be found here [19].
In order to evaluate a given specification, the same has to be well-formed, i.e. there can be no cycle in the corresponding dependency graph with a total weight of zero. This would be equivalent to the streams contained in the cycle influencing each other's new values, leading to an infinite loop and therefore making it impossible to evaluate them.
The following example is a simple, well-formed specification:

```
input in : Float32
output out_one := in + out_two
output out_two := out_one.offset(by: -2).defaults(to: 0.0)
trigger out_two + in.offset(by: -3).defaults(to: 42.0) > 50.0
```

Listing 3.8: Sample specification to demonstrate dependency graph

We now take a look at the dependency graph of this example, where the nodes correspond to the streams with the same names and the trigger is labeled as `t1`.
As is fairly obvious by the graph in Figure 3.1 for the example from Listing 3.8, there is exactly one cycle between `out_one` and `out_two`, but since the sum of the values on the edges is not equal to 0 it does not endanger our criterion for well-formed specifications.

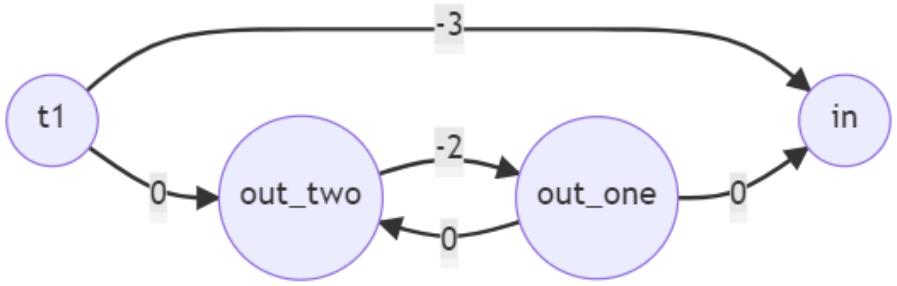


Figure 3.1: Dependency graph for the specification in Listing 3.8

3.2 RTLola Frontend

A specification in RTLola is analyzed in multiple stages: First, by means of a grammar, an abstract syntax tree (AST) is generated.
This AST is then processed through various stages to produce a high-level intermediate represenatation (HIR) of the specification that contains additional information from the semantic analysis that is used to perform various transformations of expressions.
The stages of the existing RTLola compiler frontend are:

- Base Mode (starting point generated from AST)
- Typed Mode (performs type analysis on the streams)
- Dependency Analysis Mode (performs dependency analysis on the streams)
- Ordered Mode (determines the evaluation order of the streams)
- Memory Bound Mode (calculates the memory bound for every stream)
- Complete Mode (final HIR)

The HIR is then lowered to a mid-level intermediate representation (MIR), stripping away unnecessary information for interpreting. This MIR is what the various backends work with.

Chapter 4

Approach

In this chapter, we will lay out our approach to introducing modularization into the RTLola language. We will discuss the structure of the proposed modularization system and the reasoning behind the decisions that were made.

4.1 Goals

Modularization of code in RTLola is done in order to achieve three stated goals:

- Interchangeability
- Organization
- Reusability

These goals are central to a successful system of modularity that promotes clear, concise and maintainable code. They also map one to one onto our proposed components.

4.1.1 Interchangeability

The first goal, interchangeability, is what is generally thought of first when modularity of a system is mentioned: A part of the system fulfilling a certain functionality can be replaced with a different part achieving the same functionality by potentially different means, all without changing any other part of the system.
This can only be achieved if we introduce some kind of contract that guarantees that all parts that are supposed to be interchangeable are immediately and verifiably found to be interchangeable. We call this contract an 'interface' and discuss it in detail in Section 4.2.1.

4.1.2 Organization

To fulfill the second goal, organization, we want to create a unit that contains all the functionality of a single part under an addressable namespace. As mentioned previously, each of these units should state which contract they abide by, thus providing a clear and concise overview of which functionality is contained in

them. We call this unit a 'module', and the contract it fulfills is the interface it implements. We discuss modules in Section 4.2.2

4.1.3 Reusability

Lastly, we want to be able to avoid patterns where code is duplicated in order to provide similar or even unchanged functionality in multiple different places. A first step towards this is done by collecting functionality of parts that might be reused in a module, but this alone is not enough. Since that collected functionality might be used multiple times over, we still need a way to express our desire to have two or more independent copies of a module, e.g. when a modules describes a replicated subsystem within a system like multiple CPUs in a large server.
We call these independent copies 'instances'. Each module automatically provides a way to obtain a new instance of it and configure which streams to use as input to the instance. We discuss instances in Section 4.2.3

4.2 Architecture

In this section, we explain the different components of the system and their purpose. We also provide examples of how to use them.

4.2.1 Interfaces

Interfaces are the starting point for the modularization process. They concisely represent the capabilities that a module will have and are the source of truth. Yet, they don't dictate anything about the implementation of the capabilities. They further don't disallow any additional streams in the module. For example, the interface in 4.1

```
interface flightHeight {
    input altitude: Float32
    constant minHeight: Float32
    output tooLow: Bool
}
```

Listing 4.1: A basic interface with one input, constant and output requirement

declares the capability to ingest a stream of altitude measurements named `altitude` and requires the module to produce at least one Boolean output called `tooLow`.
Additionally, it declares a constant requirement for a `minHeight`, which conveys the intention to make the calculation of `tooLow` based on the value of `minHeight`. But intention is all that can be conveyed, no actual contractual obligation is expressed here.

4.2.2 Modules

A module is the central part of the modularization process. It is the place where all of the functionality for a given interface is implemented, such as the stream expressions for all the output streams.

Of special note is the fact that a module can only refer to stream names inside either itself or the input streams from the interface it implements. It shall declare all the output and constant streams from the interface, though additional ones are allowed.
In the following example, we declare a module that implements the interface from Listing 4.1.

```
module generic_flightHeight implements flightHeight {
    constant minHeight := 100.0
    output tooLow := altitude < minHeight
}
```

Listing 4.2: Definition of a module implementing the 'flightHeight' interface from Listing 4.1

In this generic example, we give the `minHeight` constant stream a value of `100.0` and use it in the computation of `tooLow` as 'intended'. But, as mentioned earlier this is not enforced, meaning the following example is a technically valid implementation of the interface:

```
module generic_flightHeight implements flightHeight {
    constant minHeight := 100.0
    output tooLow := altitude < 125.0 //Instead of the minHeight
        constant, we use a different number
}
```

Listing 4.3: Still valid example of a module implementing the 'flightHeight' interface from Listing 4.1

As long as the `minHeight` constant stream is declared, the module is valid, even if it is not used.
Finally, we can use the module level as a differentiating factor to express specialization. If we have a system with different sensors that measure the same metric but with different parameters or requirements, we can use a generic interface to create specialized modules. This is shown in the following example, where a drone uses both GPS and LIDAR sensors to determine whether or not is flying too low.

```
module gps_flightHeight implements flightHeight {
    constant minHeight := 75.0
    output tooLow := altitude < minHeight
}

module lidar_flightHeight implements flightHeight {
    constant minHeight := 50.0
    output tooLow := altitude < minHeight
}
```

Listing 4.4: Example with differently specialized modules

The two sensors have different requirements and thus need specialization, but not to the degree that they should have separate interfaces as they both perform a similar function, e.g. determining whether or not the flight height is sufficient.

4.2.3 Instances

A module by itself does not actually generate any output streams. To make the implemented functionality of the module come into existence the module has to be instantiated. The streams are then accessible under the namespace of the instance. The following example shows this by creating an instance of the module from Listing 4.3:

```
input measured_altitude: Float32
instance simple := generic_flightHeight.instantiate(altitude:
    measured_altitude)
```

Listing 4.5: Simple instance

Here we use the `measured_altitude` stream from the scope of the instance as the input `altitude` from the interface of the module, i.e. `measured_altitude` will be used in the stream expression to calculate `tooLow`. The resulting value will be available in the scope of the instantiation as `simple.tooLow`.
Instances allow for customization and duplication of finished modules. They provide a convenient way to obtain multiple independent copies of a module that use separate resources, i.e. streams. In the following example we make use of this by monitoring a redundant set of LIDAR sensors.

```
input measured_altitude_lidar_a: Float32
input measured_altitude_lidar_b: Float32
instance sensor1 := lidar_flightHeight.instantiate(altitude:
    measured_altitude_lidar_a)
instance sensor2 := lidar_flightHeight.instantiate(altitude:
    measured_altitude_lidar_b)
```

Listing 4.6: Multiple instantiation of the same module

By instantiating the module twice with different input streams we obtain separate copies that make separate streams available, i.e. `sensor1.tooLow` is different from `sensor2.tooLow`. This gives us the ability to define functionality once and reuse the implementation details without spelling them out again.
The streams are mapped to the input streams of the interface during the instantiation of the module. Optionally, the mapping can be left out, which results in the use of the stream names from the interface to look for compatible streams of the same name in the scope of the instantiation. This can lead to an even simpler instantiation than Listing 4.7:

```
input altitude: Float32
instance even_simpler := generic_flightHeight.instantiate()
```

Listing 4.7: Simple instance

Since the names, both in the scope of the instantiation and inside the module match, the mapping does not have to be specified.

Chapter 5

Implementation

In this next chapter we discuss how we implemented our approach in the existing RTLola frontend. We will explain modifying the parser to accept the new syntax. We also formalize the verification of modules and provide the algorithmic approach used in the compiler frontend to verify them.

5.1 Modifications to the parser

The existing RTLola parser had to be modified to accept the additional syntax for the module system. It is built on parsing expression grammars, so we defined appropriate rules for interfaces, modules and instances as well as changing the existing rule for declarations - the building blocks of specifications - to include our new rules.

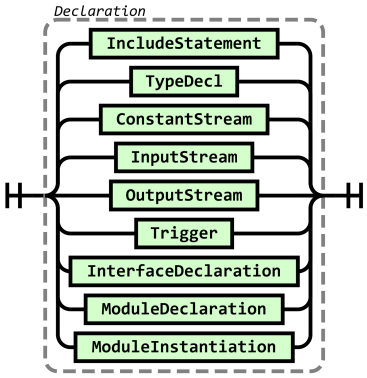


Figure 5.1: Railroad diagram for the modified declaration rule at the root of RTLola specifications

As a first step we have to change the rule that specifications are based on. They are a series of declarations, so by including our own rules as choices in the 'Declaration' rule, we make it possible to declare our own structures in a specification. The modified rule is visualized in Figure 5.1.

The `InterfaceDeclaration`, `ModuleDeclaration` and `ModuleInstantiation` rules have been added, making them usable inside a specification.

Interfaces

To make interfaces work, we have to add a few more rules. This includes the `InterfaceDeclaration` as the entry point for interfaces. It consists of the keyword 'interface' followed by an Interface name, which is just a double indirection to the already defined `Ident` and closes out with an `InterfaceBlock` contained in curly braces.
This `InterfaceBlock` consists of multiple `InterfaceStatement`s, which can be either a `InputRequirement`, `OutputRequirement` or a `ConstantRequirement`. Each of those express requirements, and therefore need an identifying keyword (input/output/constant), an identifier and a type annotation. The output requirement can also optionally declare an activation condition on the stream.
The corresponding railroad diagram is in Figure 5.2.

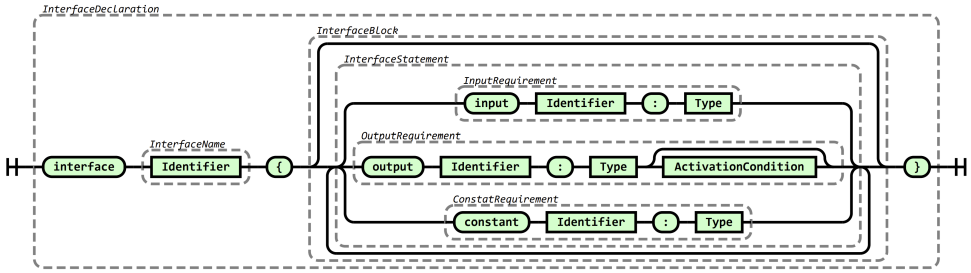


Figure 5.2: Railroad diagram for the declaration of an interface

5.1.1 Modules

Modules are a little bit simpler, since they mostly rely on existing rules.
Much like the interface, a module uses the `ModuleDeclaration` as the entry point, consisting of the keyword `module`, followed by an identifier for the module, the keyword `implements`, the identifier of the implemented interface and the curly-braces-enclosed `ModuleBlock`.
The `ModuleBlock` also consists of multiple `ModuleStatement`s, which are either type declarations, output stream declarations, trigger declarations or constant stream declarations as known from regular RTLola.
The corresponding railroad diagram is in Figure 5.3.

5.1.2 Instances

Lastly, the way to instantiate a module is to provide the module's name and after the keyword `instantiate` and assign this to a variable marked with the keyword `instance`. Optionally, a name mapping can be provided after the keyword `with` , enclosed in square brackets.
The corresponding railroad diagram is in Figure 5.4.

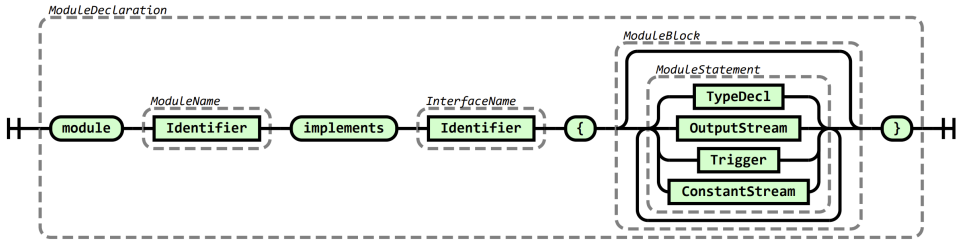


Figure 5.3: Railroad diagram for the declaration of a module

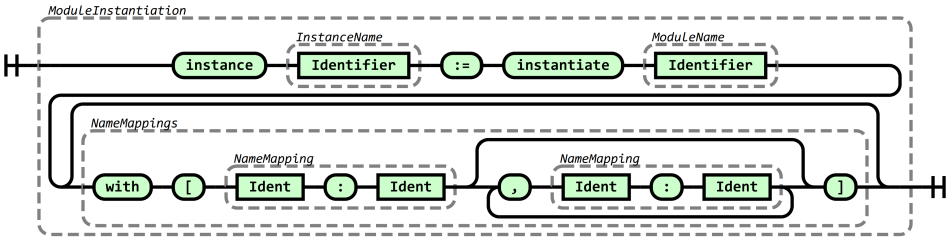


Figure 5.4: Railroad diagram for the declaration and assignment of an instance

5.2 Formal requirements for validity

Interfaces, modules and instances have a few rules that need to be checked that can't be checked by the grammar alone. To formalize these requirements, we define the following.
Let

- I be the set of tuples $(id_i, In_i, Out_i, Const_i)$ describing the interfaces with their identifier, the sets of their input requirements, their sets of output requirements and their sets of constant requirements in a specification S
- a requirement of any kind r be denoted by the tuple $(id_r, type_r)$ with the identifier of the requirement and the value type of the requirement
- R_i be the union of all the requirements for a given interface $i \in I$: $R_i = In_i \cup Out_i \cup Const_i$
- M be the set of tuples $(id_m, In_m, Out_m, Const_m, id_{i_m})$ describing the modules with their identifier, the sets of their input streams, the sets of their output streams, the sets of their constant streams and the identifier of the implemented interface in a specification S
- S_m be the union of all input, output and constant streams in a given module $m \in M$: $S_m = In_m \cup Out_m \cup Const_m$

- $i_m \in I$ be the interface that a given module $m \in M$ implements, identified by id_{i_m}
- an input stream *in* be described by the tuple $(id_{in}, type_{in})$ with the id of the stream and its value type
- an output stream *out* be described by the tuple $(id_{out}, type_{out}, Sa_{out})$ with the id of the stream, its value type and the set of the ids of the streams it accesses
- a constant stream *cons* be described by the tuple $(id_{cons}, type_{cons}, val_{cons})$ with the id of the stream, its value type and its value
- Ins be the set of tuples $(id_{ins}, id_{m_{ins}})$ describing the instances with their identifier and the identifier of the module they instantiate in a specification S
- Id be the multiset (we use square brackets to denote multisets) of identifiers used in S and Id_I be the multiset of identifiers used by all the interfaces, Id_M be the multiset of all the identifiers used by all the modules and Id_{Ins} be the multiset of all the identifiers used by all the instances in S.
  $Id_I \subset Id \wedge Id_M \subset Id \wedge Id_{Ins} \subset Id$
- m_{in} be the module instantiated by a given instance in and i_{in} be the interface implemented by m_{in}

5.2.1 Interfaces

Interfaces are the easiest to check during this stage, since none of their requirements directly rely on other declarations in the scope.
For an interface we only have to check that its identifier is not being used already in the current scope either by an interface or any other way - as per Equation 5.1 - and that there are no duplicate identifiers for requirements inside the interface, which is insured by Equation 5.2.

$$\forall i \in I : i \text{ is valid}$$

$$\iff$$

$$|[id_x \in Id \mid id_x = id_i]| = 1 \tag{5.1}$$

$$\wedge$$

$$|\{id \in R_i\}| = |\{id \in In_i\}| + |\{id \in Out_i\}| + |\{id \in Const_i\}| \tag{5.2}$$

The following example breaks all of these requirements, as the interface is reusing an identifier both ways and also requires both an input and output stream of the same name:

```
input example: Int32

interface example {}

interface example {
    input example_requirement: Bool
    output example_requirement: Bool
}
```

Listing 5.1: Specification violating all interface related requirements

5.2.2 Modules

Modules are a more involved. We have to check that:

- its identifier is not in use already, either by another module or by anything else
- the identifier of the implemented interface is in use and refers to a valid interface
- there are no duplicate identifiers inside the module
- the module fulfills all the output requirements of the interface
- the module fulfills all the constant requirements of the interface
- the module only accesses streams defined inside the module

To express these requirements formally:

$$\forall m \in M : m \text{ is valid}$$

$$\iff$$

$$|[id_x \in Id \land id_x = id_m]| = 1 \tag{5.3}$$

$$\land$$

$$id_{i_m} \in \{id_x \mid (id_x, In_x, Out_x, Const_x) \in I \land x \text{ is valid } \} \tag{5.4}$$

$$\land$$

$$\forall s \in S_m : |[id_x \mid x \in S_m \land id_x = id_s]| = 1 \tag{5.5}$$

$$\land$$

$$\forall o_{i_m} \in Out_{i_m} \exists o_m \in Out_m : id_{o_{i_m}} = id_{o_m} \land type_{o_{i_m}} = type_{o_m} \tag{5.6}$$

$$\land$$

$$\forall c_{i_m} \in Const_{i_m} \exists c_m \in Const_m : id_{c_{i_m}} = id_{c_m} \land type_{c_{i_m}} = type_{c_m} \tag{5.7}$$

$$\land$$

$$\forall o_m \in Out_m : \forall sa \in Sa_{o_m} : sa \in [id_x \mid x \in S_m] \tag{5.8}$$

Equation 5.3 formalizes the first requirement analogous to interfaces and Equation 5.1. Equation 5.4 checks the list of interfaces for the provided identifier and the validity of the interface behind it, while Equation 5.5 makes sure that there are no duplicate identifiers inside the module. Equation 5.6 and 5.7 ensure that all the requirements - output and constant respectively - are fulfilled. Lastly, Equation 5.8 states that no stream inside the module may access streams from the surrounding scope by making sure all the stream accesses are within the streams inside the module.
If these equations are all satisfied for a given module, we consider it a valid module. Again, we provide examples that break the above requirements:

```
input example: Float32
input not_example_interface: Bool

interface example_interface {
    input example_in: Float32
    output example_out: Bool
}

module example implements not_example_interface {
    output example_out := example_in > 42.0
}
```

Listing 5.2: Specification violating Equations 5.3 and 5.4

This first example breaks the Equation 5.3 in line 9, by having the module use the same identifier as the above declared input stream.
It also breaks the second Equation 5.4 in the same line, as the module implements `not_example_interface`, which - though it is in use - does not refer to a valid interface.

```
input example: Float32

interface example_interface {
    input example_in: Float32
    output example_out: Bool
}

module example_module implements example_interface {
    output example_out := example_in > 42.0
    output example_out := example
}
```

Listing 5.3: Specification violating Equations 5.5, 5.6 and 5.8

The second example breaks Equations 5.5, 5.6 and 5.8, because we have a duplicate use of the `example_out` identifier and because the latter use is both of the wrong type and also referencing a stream from the surrounding scope of the module.

5.2.3 Instances

Once an instance of a module is created, there are a few things left to ensure:

- The identifier of the instance is not in use already, either by another instance or by anything else
- The identifier of the instantiated module is in use and refers to a valid module
- None of the identifiers that the instances streams will use are in use already before the unrolling takes place
- For every input stream of the module, there needs to exist an input or output stream in the scope that the instance was declared in

The last requirement is changed, if the instance declaration comes with a mapping. Since mappings allow streams to be renamed, we can use input or output streams that don't match the specified name inside the module by providing a mapping that maps the new, chosen name to the name specified inside the module.

$$\forall in \in In : in \text{ is valid}$$

$$\iff$$

$$|[id_x \in Id \land id_x = id_{in}]| = 1 \tag{5.9}$$

$$\land$$

$$id_{m_{in}} \in \{id_x \mid (id_x, In_x, Out_x, Const_x, id_{i_m}) \in M \land x \text{ is valid } \} \tag{5.10}$$

$$\land$$

$$\forall o \in out_{m_{in}} : \text{`}id_{in}.id_o\text{`} \notin Id \tag{5.11}$$

$$\land$$

$$\forall i \in in_{i_{in}} : \exists id_x \in Id : id_x = id_i \land type_x = type_i \tag{5.12}$$

It is important to note that Equation 5.11 can only hold before code transformations for the unrolling have taken place, since after the transformations the instantiated streams will have been added.
Again, Equation 5.9 are analogous to Equations 5.1 and 5.3 for identifier availability, and 5.10 is analogous to Equation 5.4, checking the module behind the identifier exists and is valid.
As noted, 5.11 ensures the names generated by combining the name of the instance and the name of the stream to form a unique name for the unrolled code has not already been taken. This can of course only hold before the unrolling transformations have taken place.
Lastly, 5.12 checks for the existence of the input requirements of the implemented interface of the instantiated module. This can only be done at the instantiation, because a module may be instantiated in a different scope than it was declared.
Lastly, let us look at two examples that break the equations unique to instances:

```
input example: Float32
input example_instance.example_out: Bool

interface example_interface {
    input example_in: Float32
    output example_out: Bool
}

module example_module implements example_interface {
    output example_out := example_in > 42.0
}

instance example_instance := example_module.instantiate(example_in:
    example)
```

Listing 5.4: Specification violating Equation 5.11

This example, though otherwise valid, violates Equation 5.11. This is because the stream identifier `example_instance.example_out`, which will be used by the unrolled stream from the instance is already in use by the input stream of the same name.

```
input example: Float32

interface example_interface {
    input example_in: Float32
    output example_out: Bool
}

module example_module implements example_interface {
    output example_out := example_in > 42.0
}
instance example_instance := example_module.instantiate(example_in:
    example_not)
```

Listing 5.5: Specification violating Equation 5.12

This last example breaks Equation 5.12, as the stream `example_not` does not exist and can therefore not fulfill the input requirement of `example_in`.

5.3 Algorithmic verification of validity

In this Section we go into how the formal requirements from Section 5.2 are algorithmically checked and - once that check is complete - how the specification is transformed to remove the module system syntax during the compilation of a specification. This algorithmic approach was implemented in the RTLola compiler frontend in Rust, but is presented here as concise pesudo-code.
We check the validity by performing a walk across the AST generated by the parser. The first step in checking the validity is ensuring the identifiers used for any interface, module or instance are unique and also don't collide with any keywords. This same check is also being done for any other definitions in the specification like output streams. Shown here in Figure 5.3, we define a method to check identifiers using a scoped declaration store. Should a duplicate identifier or a collision with a keyword occur, it will return an error. If the identifier is ok, it is added to the scoped declaration store.
Then we need to call this identifier check on all the interfaces, modules and instances in our specification during the general check of a specification, as shown in Figure 5.6. The next step in checking validity is to check the definitions of interfaces, modules and instances. For interface and module definitions we have to recurse into their bodies so we can also check there for duplicate and colliding identifiers. We also have to make sure that the stream expressions for output streams in modules are correct.
We go into greater detail what it formally means for interfaces, modules and instances to be correct in Section 5.2. The RTLola paper [19] also goes into greater detail on when a stream expression is valid.
This is shown in Figure 5.7, where we start with checking the interfaces by iterating over all of them, pushing a new scope to the declaration store and performing the same duplicate and collision check from before inside the interface to make sure no requirements are identically named or collide with keywords. Before leaving, we pop the interfaces scope back off the current scope. By introducing a new scope at the start of the check and discarding it at the end we ensure that we only check for duplicates within the interface.

```
func check_identifier(declaration) {
    if keyword_list contains declaration.identifier:
        throw keywordCollisionError

    if current_scope.declaration_store contains declaration.identifier:
        throw duplicateIdentifierError
    else:
        current_scope.declaration_store += declaration.identifier
}
```

Figure 5.5: Checking the identifiers for a given declaration

```
func check(specification) {
    ...

    for interface in specification.interfaces:
        check_identifier(interface)

    for module in specification.modules:
        check_identifier(module)

    for instance in specification.instances:
        check_identifier(instance)

    ...

    check_interfaces(specification.interfaces)
    check_modules(specification.modules)
    check_instances(specification.instances)

    ...
}
```

Figure 5.6: Checking the identifiers of interfaces, modules and instances as well as checking them for valid definitions

```
func check_interfaces(interfaces) {
    for interface in interfaces:
        current_scope.push

        for input_requirement in interface.input_requirements:
            check_identifier(input_requirement)
        for output_requirement in interface.output_requirements:
            check_identifier(output_requirement)
        for constant_requirement in interface.constant_requirements:
            check_identifier(constant_requirement)

        current_scope.pop
}
```

Figure 5.7: Checking interface body

```
func check_modules(modules) {
    for module in modules:
        if !module.implemented_interface.isValid:
            throw implementedInterfaceInvalidError

        current_scope.push

        for output in module.outputs:
            check_identifier(output)
            check_expression(output)
        for constant in module.constants:
            check_identifier(constant)

        for output_requirement in module.interface.output_requirements:
            if output_requirement is not fulfilled:
                throw OutputRequirementNotFulfilledError

        for constant_requirement in module.interface.constant_requirements:
            if constant_requirement is not fulfilled:
                throw ConstantRequirementNotFulfilledError

        current_scope.pop
}
```

Figure 5.8: Checking module body

Next, we perform a similar check on the modules. Shown in Figure 5.8, we begin by checking whether or not the interface that the module implements is defined and valid in the surrounding scope. We then push a new scope for the module. This is necessary, since a module can introduce additional streams not required by the interface, which also again have to be checked. Additionally, we check that the stream expressions for all output streams are valid. Lastly, we check whether or not all output and constant requirements from the interface have been fulfilled, meaning there is a stream of the same type and name available. Last, but not least, we check the instantiations. To make sure an instantiation is valid, we first check if the module being instantiated is valid. Subsequently, we make sure that all the input streams are available. If there is a name mapping, we consider it during the lookup of the input streams.
After all the checks have completed, we add the output streams to the specification. Their name gets replaced by a namespaced identifer generated from the name of the instance, and the surrounding scope is checked again for duplicates with that namespaced identifier.

```
func check_instances(instances) {
    for instance in instances:
        if !instance.instantiated_module.isValid:
            throw instantiatedModuleInvalidError

        for input_requirement in instance.module.interface.input_requirements:
            if instance.nameMapping exists:
                if input_requirement is not fulfilled with mapping:
                    throw InputRequirementNotFulfilledError
            else:
                if input_requirement is not fulfilled:
                    throw InputRequirementNotFulfilledError

        for output in instance.module:
            output.name = "${instance.name}.${output.name}"
            check_identifier(output)
            specifications.outputs.add(output)
}
```

Figure 5.9: Checking module body

Chapter 6

Evaluation

In this chapter, we evaluate our implementation from Section 5 by providing a refactored specification from another RTLola thesis to quantify our claims of reduced duplication and analyze the results of a case study. This study is structured as a questionnaire given to professional users of the RTLola language that aims to answer a set of qualitative research questions described in the following section.

6.1 Research questions

The questions are assorted in three different categories according to the aspect of the module system they align with and the goal from Section 4.1 they aim to fulfill. The aspects we want to study are the ease of use (6.1) as it relates to the effort spent by the programmer on modularizing the code, the power and expressiveness of the language - specifically the module system - (6.1) and finally the effects the module system has on the code written (6.1).

Ease of use

The question of whether or not a certain tool is easy to use is highly subjective to the individual using said tool. As such, it is not always easy to establish good criteria for ease of use. It is however an important quantifier for how likely people are to adopt a tool, as generally easier tools to use provide a greater benefit to the user.
In the case of our module system, ease of use can be understood as a function of the amount of effort a user had to invest to fully grasp how the system works and is intended to be used.
This can be further subdivided into two parts: firstly understanding all the components that make up the module system - interfaces, modules and instances on their own - and secondly understanding how they perform together to create a coherent unit.

Power and Expressiveness

The modularization system is a compromise between being able to easily implement it and being able to express complex systems and structures. As such, we

need to evaluate if the current expressiveness is sufficient to codify the needed complexity of specifications.

Effects on specifications

Questions in this category are mostly concerned with whether or not the modularization system will have an impact on the way users write specifications. More specifically, we want to find out whether or not they write less or more complex specifications, whether or not readability increases and if it can help avoid duplication.

6.2 Analysis

In this section, we present the results of the questionnaire, which can be found in full in the appendix B.

Questionnaire results

Active participants were given 10 statements and asked to rate their agreement/disagreement on a fluid marking scale as well as 5 free text questions. They filled out the questionnaire after completing a short exercise in RTLola, being given a presentation to introduce the modularization system and finally completing the same task with modules to assess their opinions on the previously asked research question categories from Section 6.1.
The statements are structured as follows:

Statements - Ease of use

- Statement 1: 'The modularization concept was easy to understand.'
- Statement 2: 'I have understood that interfaces function as a contract that modules have to abide by.'
- Statement 3: 'I have understood that modules are akin to a blueprint for a unit that contains related functionality.'
- Statement 4: 'I have understood that instances are individual and independent copies of the unit that was laid out in the instantiated module.'
- Statement 5: 'It is unclear to me how interfaces, modules and instances work together.'

Statements - Effects on specifications

- Statement 6: 'The modularization approach will not make specifications more concise and organized.'
- Statement 7: 'The modularization approach will lead to less complexity in specifications I write.'

Statements - Power and Expressiveness

- Statement 8: 'The implementation of modules does not align well with my existing specifications.'
- Statement 9: 'I already use techniques (e.g. strict naming rules for streams) to organize my code.'
- Statement 10: 'I already make sure to give my specifications a lot of structure.'

The statements given to the five participants were either formulated positively or negatively and they are marked as such accordingly in the table. We normalized the results, meaning the score has been inverted if the statement was phrased negatively, resulting in a agreement score with the positive formulation.
From this data we suggest that the modularization system is quite welcome with the surveyed users. Clearly, on most questions there is a strong positive agreement with the module system in its current state. Only statements 5, 6 and 8 seem controversial when looking at the average score plotted in Figure 6.1. This is noteworthy, because those are the exact statements that are negatively formulated. A table with the dataset is available in the appendix A.
Looking more closely at the individual scores in Figure 6.2 seems to suggest

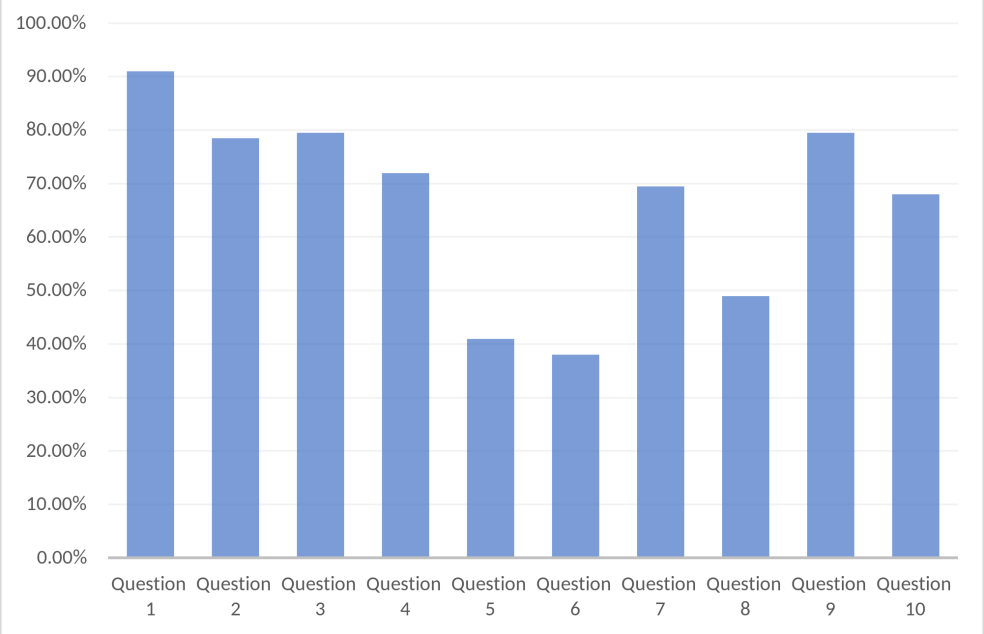


Figure 6.1: Average agreement score (normalized) per statement, in percent

participant error in understanding the statement though, since all their peers answered with a similar score but in the opposite direction. Since it is always a different person in each statement and they are the only and extreme outlier, we can assume they misunderstood when switching to a negatively formulated

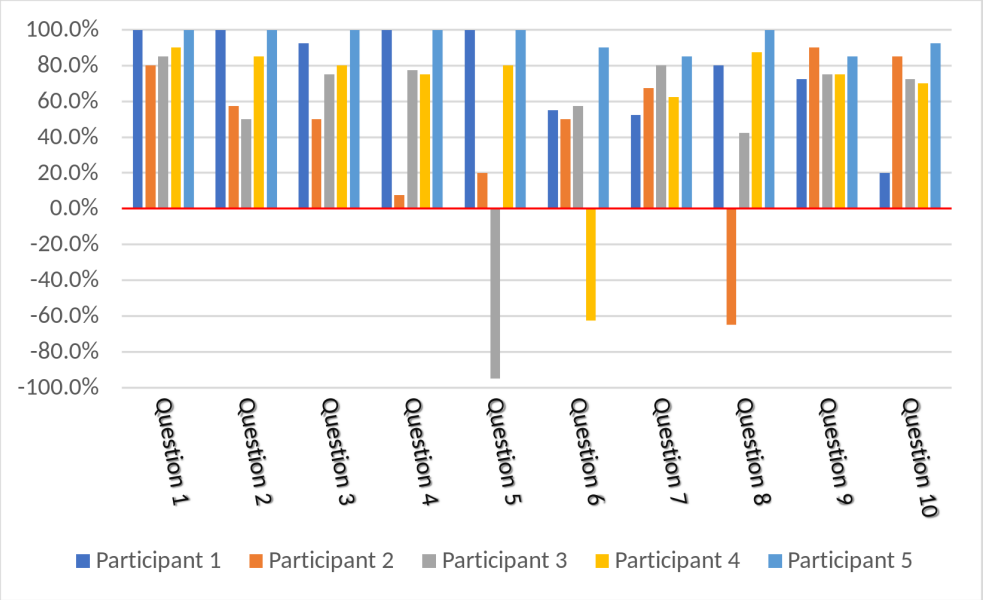


Figure 6.2: Per-person agreement score per statement (normalized), in percent

statement and filled the questionnaire out incorrectly as a result.

On all of the positively formulated statements the spread is very close - as can be seen in Figure 6.3 - and leans towards strong agreement with the statements. From this we conclude that the users are sufficiently satisfied with the general implementation and turn towards the free text questions for further detail on their opinions and suggestions for improvement.

Free text questions

The previous conclusion is echoed by the free-text feedback. All users stated that the modularization system would make it easier to write complex specifications and reduce the difficulty and effort needed in writing repetitive functionality. Beyond that, 3 main suggestions for improvement were made, all by at least 2 participants independently:

- Changing constants such that they are provided with a default value in the module declaration that can optionally be overwritten during the instantiation
- Allowing modules to implement multiple interfaces at the same time
- Templating for types in interfaces/modules

Though out of scope for implementation during this thesis, all 3 of these would be positive additions and will be further discussed in chapter 7 on future work.

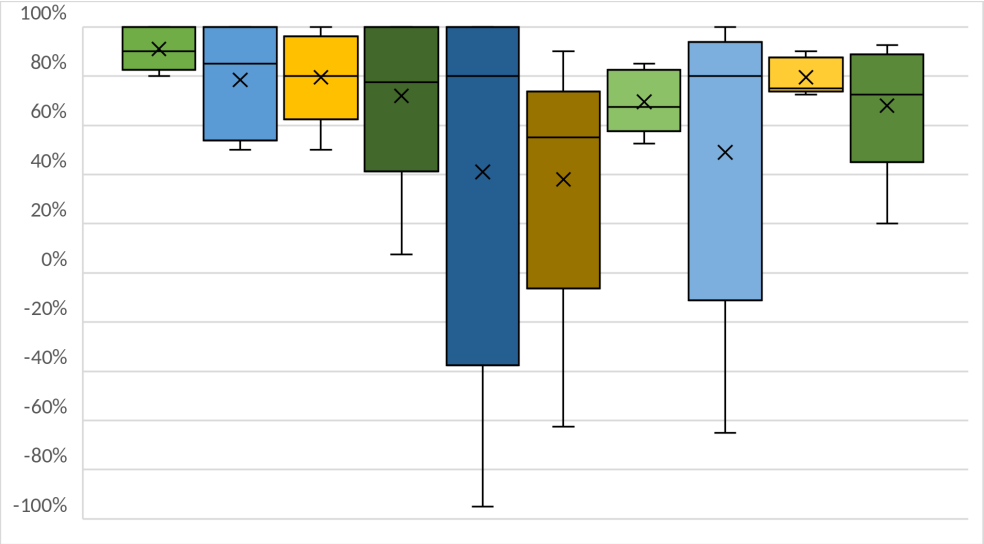


Figure 6.3: Agreement score per statement (normalized), in percent, Statement 1 to 10 from left to right

6.3 Refactored specification

In the master thesis from Baumeister [2], they provide an example specification in their evaluation that monitors a UAVs position to check whether or not it stays within a granted flight area. This is realized by checking if the current coordinates of the UAV are within a given polygon, called a geofence.
The original specification for a 12 line polygon was 94 lines, with each line in the polygon consisting of 4 output streams. They each use a small set of latitude and longitude calculating streams defined in the beginning of the specification. With minimal effort, we were able to reduce this to a mere 72 lines (22.6% reduction), with each new line in the polygon now only needing another instance of the `line_module`. This also vastly reduces duplicated stream expressions, reducing the chances of bugs. Both specifications can be found in full in the appendix respectively C.1 and C.2.

Chapter 7

Future work

In this chapter, we discuss possible future enhancements to the modularization system that were out of scope for this thesis, including the suggestions made by participants of the evaluation.

Constant overwrite during Instantiation

The most frequently heard suggestion was a change to the way constants are being handled. In addition to providing a value for a constant when writing the module, additional value could be gained from treating the value provided in the module as a default value that can be overwritten during the instantiation process.
That way the process of creating more unique instances of the same module, reducing the need to write separate modules just in case a constant differs. For example, during the task in the questionnaire from Section 3.1.8, one could create a general room module and then just pass in the required temperatures at instantiation time.

Multiple implemented interfaces per module

Another often suggested improvement is allowing modules to implement multiple interfaces. This would allow for smaller units of functionality to be combined to yield a larger set of capabilities.
For example, the bathroom from the questionnaire task could be composed out of a heater control interface and humidity control interface.

Type templating

Lastly, some participants suggested a templating system for types inside interfaces. That way, an interface can be kept even more generic by giving the module concrete control over typing. Instead of concrete types like `Int64` or `Bool`, placeholders would be introduced to place constraints on sets of streams. They would then be resolved when defining the module, e.g. as shown in Listing 7.1.

```
interface generic_example {
    input gen_in: <S>
    output gen_out: <S>
    output gen_out2: <T>
}

module specific_module implements generic_example<S:Int64, T:Bool> {
    output gen_out := gen_in + 10
    output gen_out2 := gen_in < 100
}
```

Listing 7.1: Example of a possible generic interface with type templating

'Higher order' modules

In functional programming, higher order functions are functions that either return a function themselves, accept a function as a parameter or perform both options. The `map` function in various programming languages is a good example: It often receives a function and a collection of elements as its parameters, applies the function to each element of the original collection and returns the collection of results.
Similarly, one could imagine use cases for both modules that accept other modules as input, allowing the user to compose the functionality of modules, as well as modules that provide other modules as part of their namespace to allow nested organization and the combination of multiple modules in a more complex structure than possible with the results of this thesis.

Binary imports

Now that we have a good way to organize and structure specifications, a last limitation on composing them should be lifted: Combining multiple documentations or parts of them together by constructing a binary import system. Such a system would go hand in hand in hand with modules.
The existing import system in RTLola to enable libraries like `math` would have to be reworked, and an analysis for cyclic dependencies would need to be implemented.

Optimizations to the specification analysis

The architecture of the module system allows for plenty of optimizations to be made during the analysis stage of the compiler frontend. One example of this is reusing previously made analysis work on modules when they are instantiated multiple times or not analysing modules beyond syntactic correctness when they are not in use. This holds lots of potential to optimize the performance of the compiler frontend in the future.

Chapter 8

Conclusion

This thesis implements a modularization system for the stream based specification language RTLola. We presented our architecture for the modularization system along with formalized verification criteria and an algorithmic approach to checking the validity of specifications containing interfaces, modules and/or instances and transforming away the module system syntax to reduce the specification back to plain RTLola.
To begin with, we laid out our architecture: It consists of three tiers of abstractions, each with a specific goal in mind when modularizing a unit of functionality. Interfaces, as a first tier, are contracts to clearly state what the unit is capable of by declaring which streams it consumes and which streams it produces. They are the source of truth for streams and their respective types, but contain no implementation details. This keeps components interchangeable, as long as they implement the same interface. Modules, as the second tier of abstraction, are the place for the specific implementation details. This keeps the unit of functionality organized by keeping all of the implementation in one context. Lastly, instances are the final tier of abstraction. They enable easy and quick reusability to prevent code duplication. Every instance of a module is an independent copy of the blueprint laid out in the module and enables the functionality specified there under its own namespace for easy access.
Next, we showed how the new constructs in the language can be formally verified and gave equations to satisfy in order for them to be valid. We also provided an algorithmic approach to verifiy those and implemented that approach in the existing RTLola compiler frontend. We made use of the fact that our new constructs can be transpiled back down to regular RTLola and introduced a transpilation step before the rest of the compiler frontend performs its analysis of the specification to leverage the existing compiler.
Lastly, we evaluated our implementation both with a large example specification to show how it can reduce complexity, size of specifications and prevent code duplication as well as a small study with users of RTLola to gauge how satisfied they were with our results. We analyzed their opinions and discussed their feedback, incorporating their suggestions in the discussion of the next steps.

Bibliography

[1] T.O. Barnett and L.L. Constantine. *Modular Programming: Proceedings of a National Symposium*. Information & systems Institute, 1968. URL: https://books.google.de/books?id=eI8%5C_HQAACAAJ.

[2] Baumeister. "Tracing Correctness: A Practical Approach to Traceable Runtime Monitoring". Master Thesis. Saarland University, 2020.

[3] Corrado Böhm and Giuseppe Jacopini. "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules". In: *Commun. ACM* 9.5 (May 1966), pp. 366–371. ISSN: 0001-0782. DOI: 10.1145/355592.365646. URL: https://doi.org/10.1145/355592.365646.

[4] F. Boussinot and R. de Simone. "The ESTEREL language". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1293–1304. DOI: 10.1109/5.97299.

[5] P. Caspi et al. "LUSTRE: A Declarative Language for Real-Time Programming". In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '87. Munich, West Germany: Association for Computing Machinery, 1987, pp. 178–188. ISBN: 0897912152. DOI: 10.1145/41625.41641. URL: https://doi.org/10.1145/41625.41641.

[6] R. Conway and D. Gries. *Primer on Structured Programming Using PL/I, PL/C and PL/CT*. Winthrop computer systems series. Little, Brown, 1976. ISBN: 9780316154253. URL: https://books.google.de/books?id=moShPQAACAAJ.

[7] Ben D'Angelo et al. "Lola: Runtime Monitoring of Synchronous Systems". In: *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*. Burlington, Vermont: IEEE Computer Society Press, June 2005, pp. 166–174.

[8] Edsger W. Dijkstra. "Chapter I: Notes on Structured Programming". In: *Structured Programming*. GBR: Academic Press Ltd., 1972, pp. 1–82. ISBN: 0122005503.

[9] Edsger Wybe Dijkstra. "On the role of scientific thought". In: *Selected writings on Computing: A Personal Perspective* (1982), pp. 60–66.

[10] Brendan Eich. "ECMAScript language specification". In: (1995). URL: https://262.ecma-international.org/12.0/#sec-modules.

[11] Peter Faymonville et al. "Real-time Stream-based Monitoring". In: *arXiv e-prints*, arXiv:1711.03829 (Nov. 2017), arXiv:1711.03829. arXiv: 1711.03829 [cs.LO].

[12] Anders Hejlsberg. "C# language specification". In: (2000). URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/namespaces.

[13] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language". In: *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018), pp. 1–1315. DOI: 10.1109/IEEESTD.2018.8299595.

[14] "IEEE Standard VHDL Language Reference Manual - Redline". In: *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002) - Redline* (2009), pp. 1–620.

[15] Daniel Jackson. "Alloy: A Language and Tool for Exploring Software Designs". In: *Commun. ACM* 62.9 (Aug. 2019), pp. 66–76. ISSN: 0001-0782. DOI: 10.1145/3338843. URL: https://doi.org/10.1145/3338843.

[16] Robert Griesemer Ken Thompson Rob Pike. "The go language specification". In: (2009). URL: https://go.dev/ref/spec#Packages.

[17] David MacQueen. "Modules for Standard ML". In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. LFP '84. Austin, Texas, USA: Association for Computing Machinery, 1984, pp. 198–207. ISBN: 0897911423. DOI: 10.1145/800055.802036. URL: https://doi.org/10.1145/800055.802036.

[18] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules". In: *Commun. ACM* 15.12 (Dec. 1972), pp. 1053–1058. ISSN: 0001-0782. DOI: 10.1145/361598.361623. URL: https://doi.org/10.1145/361598.361623.

[19] Maximilian Schwenger. "Lets not Trust Experience Blindly: Formal Monitoring of Humans and other CPS". Master Thesis. Saarland University, 2019.

[20] Robert W. Sebesta. *Concepts of programming languages*, pp. 22–24. ISBN: 978-0-13-139531-2.

[21] H. Sutter and A. Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. C++ In-Depth Series. Pearson Education, 2004. ISBN: 9780132654425. URL: https://books.google.de/books?id=mmjVIC6WolgC.

[22] "The Rust reference". In: (2022). URL: https://doc.rust-lang.org/stable/reference/items/modules.html.

[23] Niklaus Wirth. "MODULA: a language for modular multiprogramming". In: *Software: Practice and Experience* 7 (Jan. 1977), pp. 1–35. DOI: 10.1002/spe.4380070102.

Appendix A

Table with normalized data from the evaluation

This table contains the results from the questionnaire statements. The results have been normalized, meaning where a negatively formulated statement was given, the resulting scores have been multiplied by -1.

#	P1	P2	P3	P4	P5	Formulation	$\emptyset$	σ
1	100%	80%	85%	90%	100%	Positive	91.0%	8.00%
2	100%	58%	50%	85%	100%	Positive	78.5%	21.07%
3	93%	50%	75%	80%	100%	Positive	79.5%	17.20%
4	100%	8%	78%	75%	100%	Positive	72.0%	33.96%
5	100%	20%	-95%	80%	100%	Negative	41.0%	74.05%
6	55%	50%	58%	-63%	90%	Negative	38.0%	52.19%
7	53%	68%	80%	63%	85%	Positive	69.5%	11.77%
8	80%	-65%	43%	88%	100%	Negative	49.0%	60.14%
9	73%	90%	75%	75%	85%	Positive	79.5%	6.78%
10	20%	85%	73%	70%	93%	Positive	68.0%	25.37%

Figure A.1: Normalized data from evaluation

Appendix B

Questionnaire for evaluation

This is the questionnaire that was given to participants in the case study.

Questionnaire

Programming task

Write a specification to manage a residential home equipped with smart home features.
The house consists of 5 rooms: A kitchen, a bathroom, a combined living/-dining room, an office and a bedroom. Each room is equipped with the usual smart-home equipment; a heater control with a temperature and humidity sensor. Additionally, the bedroom as well as the living/dining room are equipped with a humidifier that can be remotely controlled. The kitchen and bathroom on the other hand have ventilation systems that can be controlled remotely and the kitchens cooking range reports the setting of all the heating elements.
The humidity in the bathroom should be kept under 70% by turning on the ventilation system and emit a warning once it stays over 80% for more than 1 hours depsite the ventilation system working. The temperature should be kept at at least 22 degrees celsius.
A temperature average of over 26 degrees celsius over the last hour should result in a warning.
The ventilation system in the kitchen should turn on once any of the cooking fields is at more than 40% power. The temperature should be kept at at least 20 degrees celsius.
A temperature average of over 24 degrees celsius over the last hour should result in a warning.
The bedroom and living/dining room should be kept at at least 50% humdity and 21 degrees celsius, but a hummdity over 65% or a temperature over 24 degrees celsius over the last hour should lead to a warning.
Lastly, the temperature in the office should be kept at at least 20 degrees with a warning if it exceeds and average of 24 degrees over the last hour.
Complete the following specification to monitor and control the home:

```
input humidity_kitchen : Float32 // Reports the relative humidity of
    the room, between 0.0 and 100.0 percent.
input temperature_kitchen : Float32 // Reports the temperature of the
    room in degrees celsius
input cooking_field_1 : Int32 // Reports percentage of power that the
    cooking field is set to, between 0 and 100
input cooking_field_2 : Int32
input cooking_field_3 : Int32
input cooking_field_4 : Int32

input humidity_bathroom : Float32
input temperature_bathroom : Float32

input humidity_livingdiningroom : Float32
input temperature_livingdiningroom : Float32
```

```
input humidity_bedroom : Float32
input temperature_bedroom : Float32

input humidity_office : Float32
input temperature_office : Float32

// ... Your expression here (or in the attached exercise.lola file):









// To activate any of the heaters/humdifiers/ventilation systems
    output ‘true‘ on the corresponding output stream
output activate_kitchen_heater : Bool := // ...your expression here
output activate_livingdiningroom_heater : Bool := // ...your
    expression here
output activate_bedroom_heater : Bool := // ...your expression here
output activate_bathroom_heater : Bool := // ...your expression here
output activate_office_heater : Bool := // ...your expression here
output activate_bedroom_humidifier : Bool := //... your expression
    here
output activate_livingdiningroom_humidifier : Bool := //... your
    expression here
output activate_kitchen_ventilation : Bool := //... your expression
    here
output activate_bathroom_ventilation : Bool := //... your expression
    here
```

Listing 1: Specification to monitor a smart home

Programming task - with modules

Use modules as described in the presentation/approach chapter of the paper to refactor your code from the previous task.

```
input humidity_kitchen : Float32 // Reports the relative humidity of
    the room, between 0.0 and 100.0 percent.
input temperature_kitchen : Float32 // Reports the temperature of the
    room in degrees celsius
input cooking_field_1 : Int32 // Reports percentage of power that the
    cooking field is set to, between 0 and 100
input cooking_field_2 : Int32
input cooking_field_3 : Int32
input cooking_field_4 : Int32

input humidity_bathroom : Float32
input temperature_bathroom : Float32

input humidity_livingdiningroom : Float32
input temperature_livingdiningroom : Float32

input humidity_bedroom : Float32
input temperature_bedroom : Float32

input humidity_office : Float32
input temperature_office : Float32

// ... Your expression here (or in the attached exercise-modules.lola
    file):
```

```
// To activate any of the heaters/humdifiers/ventilation systems
    output ‘true‘ on the corresponding output stream
output activate_kitchen_heater : Bool := // ...your expression here
output activate_livingdiningroom_heater : Bool := // ...your
    expression here
output activate_bedroom_heater : Bool := // ...your expression here
output activate_bathroom_heater : Bool := // ...your expression here
output activate_office_heater : Bool := // ...your expression here
output activate_bedroom_humidifier : Bool := //... your expression
    here
output activate_livingdiningroom_humidifier : Bool := //... your
    expression here
output activate_kitchen_ventilation : Bool := //... your expression
    here
output activate_bathroom_ventilation : Bool := //... your expression
    here
```

Listing 2: Specification to monitor a smart home with modules

Questions

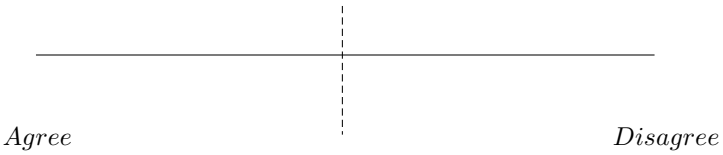
The rating questions are on a fluid marking scale. The middle is marked with a dotted line and represents a neutral answer to the question/statement. Either side is labeled either 'Agree' or 'Disagree', please mark in your answer correspondingly.

Rating Questions

Understanding the concepts

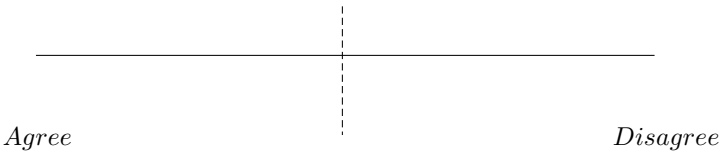
Question 1:

- The modularization concept was easy to understand.



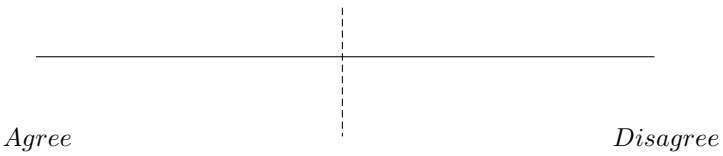
Question 2:

- I have understood that interfaces function as a contract that modules have to abide by.



Question 3:

- I have understood that modules are akin to a blueprint for a unit that contains related functionality.



Question 4:

- I have understood that instances are individual and independent copies of the unit that was laid out in the instantiated module.

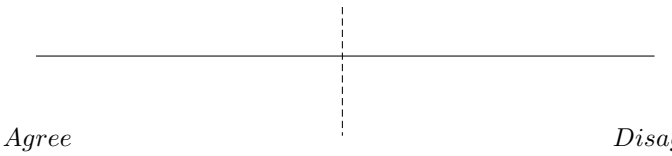
Question 5:

- It is unclear to me how interfaces, modules and instances work together.

Benefits provided by the concept

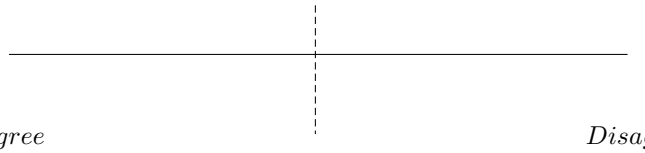
Question 6:

- The modularization approach will not make specifcations more concise and organized.



Question 7:

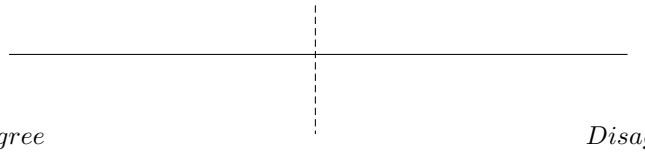
- The modularization approach will lead to less complexity in specifications I write.



Alignment with existing specifications/skills

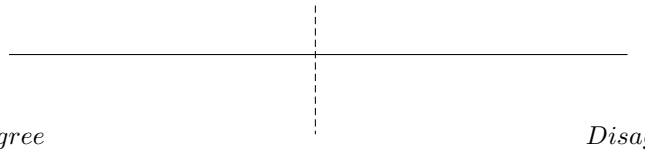
Question 8:

- The implementation of modules does not align well with my existing specifications.



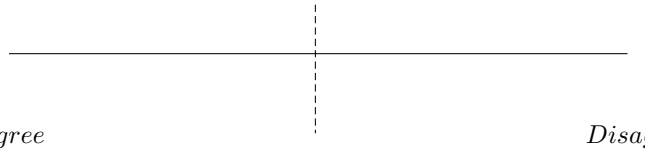
Question 9:

- I already use techniques (e.g. strict naming rules for streams) to organize my code.



Question 10:

- I already make sure to give my specifications a lot of structure.



Free Text Questions

- Is there any aspect(s) of the modularization system that you particularly dislike, would like to see changed and or removed?
- Is there any aspect(s) of the modularization system that you particularly like and/or would like to see expanded?
- Is there any functionality/functionalities you are missing in the modularization system?
- Are there any concepts in the modularization system that remain unclear to you?
- Do you have any remaining feedback on the introduction of modularization to RTLola that was not covered by previous questions?

Appendix C

Geofence specifications

This is both the original geofence specification from Baumeister [2] as well as a refactored version we created with the new module system. We also adopt the usage of constant overwriting as suggested per 7.

```
import math
input lat_in_degree : Float32
input lon_in_degree : Float32
output lat := lat_in_degre * 3.14159265359 / 180.0
output lon := lon_degree * 3.14159265359 / 180.0
output lat_pre := lat.offset(by: -1).defaults(to: lat)
output delta_lat := lat - lat_pre
output min_lat := if lat < lat_pre then lat else lat_pre
output max_lat := if lat > lat_pre then lat else lat_pre
output lon_pre := lon.offset(by: -1).defaults(to: lon)
output delta_lon := lon - lon_pre
output min_lon := if lon < lon_pre then lon else lon_pre
output max_lon := if lon > lon_pre then lon else lon_pre
output isFnc := abs(delta_lat) > 0.00000001
output m := if isFnc then (delta_lon) / (delta_lat) else 0.0
output b := if isFnc then lon-(m*lat) else 0.0

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

// Polygonline p1p2: (0.11908193655673749, 0.04525405540696442) to
    (0.11617135825882872,0.046935706350154524)
output intersect_p1p2 := abs(m--0.5777721026774507) > 0.00000001
output lat_p1p2 := if isFnc and intersect_p1p2 then (b -
    0.11405627628225343) / (-0.5777721026774507 - m) else lat
```

```
output lon_p1p2 := -0.5777721026774507 * lat_p1p2 +
    0.11405627628225343
output check_p1p2 := (intersect_p1p2) and ((lat_p1p2 > min_lat and
    lat_p1p2 < max_lat) and (lon_p1p2 > min_lon & lon_p1p2 <
    max_lon)) and ((lat_p1p2 > 0.11617135825882872 and lat_p1p2 <
    0.11908193655673749) and (lon_p1p2 > 0.04525405540696442 &
    lon_p1p2 < 0.046935706350154524))

// Polygonline p2p3: (0.11617135825882872, 0.046935706350154524) to
    (0.11766289020098507,0.05297933593688546)
output intersect_p2p3 := abs(m-4.051961219143263) > 0.00000001
output lat_p2p3 := if isFnc and intersect_p2p3 then (b -
    -0.42378613208981786) / (4.051961219143263 - m) else lat
output lon_p2p3 := 4.051961219143263 * lat_p2p3 + -0.42378613208981786
output check_p2p3 := (intersect_p2p3) and ((lat_p2p3 > min_lat and
    lat_p2p3 < max_lat) and (lon_p2p3 > min_lon & lon_p2p3 <
    max_lon)) and ((lat_p2p3 > 0.11617135825882872 and lat_p2p3 <
    0.11766289020098507) and (lon_p2p3 > 0.046935706350154524 &
    lon_p2p3 < 0.05297933593688546))

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

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

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

```
    max_lon)) and ((lat_p5p6 > 0.12039516145301407 and lat_p5p6 <
    0.12135284424675223) and (lon_p5p6 > 0.05413856066294947 &
    lon_p5p6 < 0.06110106989155227))

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

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

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

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

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

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

// Activates termination
output any_violated := !
    any_violated.offset(by:-1).defaults(to:false) and (height_check
    or check_p0p1 or check_p1p2 or check_p2p3 or check_p3p4 or
    check_p4p5 or check_p5p6 or check_p6p7 or check_p7p8 or
    check_p8p9 orcheck_p9p10 or check_p10p11 or check_p11p12)
output counter : Int32:= counter.offset(by:-1).defaults(to:0) + 1
output time_counter : Int32 @1Hz :=
    time_counter.offset(by:-1).defaults(to:0) + 1
```

Listing C.1: Specification to monitor UAV geofence compliance

```
import math
input lat_in_degree : Float32
input lon_in_degree : Float32
output lat := lat_in_degre * 3.14159265359 / 180.0
output lon := lon_degree * 3.14159265359 / 180.0
output lat_pre := lat.offset(by: -1).defaults(to: lat)
output delta_lat := lat - lat_pre
output min_lat := if lat < lat_pre then lat else lat_pre
output max_lat := if lat > lat_pre then lat else lat_pre
output lon_pre := lon.offset(by: -1).defaults(to: lon)
output delta_lon := lon - lon_pre
output min_lon := if lon < lon_pre then lon else lon_pre
output max_lon := if lon > lon_pre then lon else lon_pre
output isFnc := abs(delta_lat) > 0.00000001
output m := if isFnc then (delta_lon) / (delta_lat) else 0.0
output b := if isFnc then lon-(m*lat) else 0.0
```

```
interface line_interface {
    input isFnc: Bool
    input m: Float32
    input b: Float32
    input lat: Float32
    input min_lat: Float32
    input max_lat: Float32
    input lon: Float32
    input min_lon: Float32
    input max_lon: Float32

    constant point_a_lat: Float32
    constant point_a_lon: Float32
    constant point_b_lat: Float32
    constant point_b_lon: Float32
    constant incl_lat: Float32
    constant incl_lon: Float32

    output intersect: Bool
    output lat_line: Float32
    output lon_line: Float32
    output check: Bool
}

module line_module implements line_interface {
    constant point_a_lat := 0.0
    constant point_a_lon := 0.0
    constant point_b_lat := 0.0
    constant point_b_lon := 0.0
    constant incl_lat := 0.0
    constant incl_lon := 0.0

    output intersect := abs(m - incl_lat) > 0.00000001
    output lat_line := if isFnc and intersect then (b -
        -0.5676754214244288) /(incl_lat - m) else lat
    output lon_line := incl_lat * lat_line + -0.5676754214244288
    output check := (intersect) and ((lat_line > min_lat and lat_line
        < max_lat) and (lon_line > min_lon & lon_line < max_lon)) and
        ((lat_line > point_a_lat and lat_line < point_b_lat) and
        (lon_line > point_a_lon & lon_line < point_b_lon))
}

instance p0p1 := instantiate line_module with [point_a_lat:
    0.1180559967662996, point_a_lon: 0.0399734162336672, point_b_lat:
    0.11908193655673749, point_b_lon: 0.04525405540696442, incl_lat:
    5.147123859035988, incl_lon: -0.5676754214244288]
instance p1p2 := instantiate line_module with [point_a_lat:
    0.11908193655673749, point_a_lon: 0.04525405540696442,
    point_b_lat: 0.11617135825882872, point_b_lon:
    0.046935706350154524, incl_lat: -0.5777721026774507, incl_lon:
    0.11405627628225343]
instance p2p3 := instantiate line_module with [point_a_lat:
    0.11617135825882872, point_a_lon: 0.046935706350154524,
```

```
    point_b_lat: 0.11766289020098507, point_b_lon:
    0.05297933593688546, incl_lat: 4.051961219143263, incl_lon:
    -0.42378613208981786]
instance p3p4 := instantiate line_module with [point_a_lat:
    0.11766289020098507, point_a_lon: 0.05297933593688546,
    point_b_lat: 0.11463282400467073, point_b_lon:
    0.054732465100919954, incl_lat: -0.5785778430078313, incl_lon:
    0.12105647715143869]
instance p4p5 := instantiate line_module with [point_a_lat:
    0.11463282400467073, point_a_lon: 0.054732465100919954,
    point_b_lat: 0.12039516145301407, point_b_lon:
    0.06110106989155227, incl_lat: 1.1052120511379069, incl_lon:
    -0.07196111344501288]
instance p5p6 := instantiate line_module with [point_a_lat:
    0.12039516145301407, point_a_lon: 0.06110106989155227,
    point_b_lat: 0.12135284424675223, point_b_lon:
    0.05413856066294947, incl_lat: -7.270162181180861, incl_lon:
    0.936393419484419]
instance p6p7 := instantiate line_module with [point_a_lat:
    0.12135284424675223, point_a_lon: 0.05413856066294947,
    point_b_lat: 0.12199513900152693, point_b_lon:
    0.05357314974496478, incl_lat: -0.8802982023152586, incl_lon:
    0.16096525129920902]
instance p7p8 := instantiate line_module with [point_a_lat:
    0.12199513900152693, point_a_lon: 0.05357314974496478,
    point_b_lat: 0.12272805380095549, point_b_lon:
    0.05060231541569706, incl_lat: -4.053451140001608, incl_lon:
    0.5480744850053588]
instance p8p9 := instantiate line_module with [point_a_lat:
    0.12272805380095549, point_a_lon: 0.05060231541569706,
    point_b_lat: 0.12135284424675223, point_b_lon:
    0.04598052457716757, incl_lat: 3.3607902333162474, incl_lon:
    -0.3618609291524651]
instance p9p10 := instantiate line_module with [point_a_lat:
    0.12135284424675223, point_a_lon: 0.04598052457716757,
    point_b_lat: 0.12447245465942572, point_b_lon:
    0.043556374755305015, incl_lat: -0.777068127486174, incl_lon:
    0.14027995202111265]
instance p10p11 := instantiate line_module with [point_a_lat:
    0.12447245465942572, point_a_lon: 0.043556374755305015,
    point_b_lat: 0.12175508788375872, point_b_lon:
    0.03872383447446285, incl_lat: 1.7783908760921594, incl_lon:
    -0.17780430293581267]
instance p11p12 := instantiate line_module with [point_a_lat:
    0.12175508788375872, point_a_lon: 0.03872383447446285,
    point_b_lat: 0.1180559967662996, point_b_lon: 0.0399734162336672,
    incl_lat: -0.3378077802156663, incl_lon: 0.07985365044243875]

// Activates termination
output any_violated := !
    any_violated.offset(by:-1).defaults(to:false) and (height_check
    or p0p1.check or p1p2.check or p2p3.check or p3p4.check or
    p4p5.check or p5p6.check or p6p7.check or p7p8.check or
    p8p9.check orp9p10.check or p10p11.check or p11p12.check)
```

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

Listing C.2: Specification to monitor UAV geofence compliance, refactored with modules