

StreamLAB Tutorial

Overview

- [What is StreamLAB?](#)
- [Getting Started](#)
- [Provided Tools](#)
- [RTLola Reference](#)

What is StreamLAB?

StreamLAB is a monitoring framework that builds on the stream-based specification language **RTLola**. RTLola combines formal semantics with a high degree of expressiveness. The formal semantics allows for static analysis of a specification giving static memory bounds, if possible. During runtime, StreamLAB computes output streams as defined by the specification based on provided input streams. RTLola allows for computing statistics and expressing real-time properties. Triggers, which are a special kind of output streams, can be used to notify the user when a given condition is fulfilled.

Getting Started

This tutorial is based on one of StreamLAB's intended application areas: monitoring drones. In this example, telemetry logs from [ArduPilot](#), generated with the [Software in the Loop](#) simulator, are used. You can follow along by using the provided `streamlab` binary or by using the [somewhat limited web interface](#). The example files and the current binary release can be found at www.stream-lab.eu:

- [Linux Binary](#) — please run `chmod +x streamlab` to make the file executable
- [Mac Binary](#) — please run `chmod +x streamlab` to make the file executable
- [Windows Binary](#)
- [Example Traces and Specifications](#)

The Setting

We will use StreamLAB for sensor validation. Our first goal will be to check that the number of visible GPS satellites is sufficient and that the GPS module provides position estimates at a given rate.

Checking the Number of Visible GPS Satellites

StreamLAB needs to know which *input streams* will be used and what their types are. Input streams are asynchronous, meaning that they do not have to provide a new value at the same time. (In the CSV input a '#' represents the absence of a value.) We start our specification by adding a declaration of a stream, which is provided by the GPS module and contains the number of visible satellites:

```
// number of visible GPS satellites
input gps: UInt64
```

The name of the input stream must currently match the column name in the CSV based input.

There are 32 GPS satellites so we should see at most 16 of them while flying close to the ground. We declare an *output stream*

```
number_of_satellites_too_large:
```

```
output number_of_satellites_too_large := gps > 16
```

Because the output stream accesses only the `gps` stream, StreamLAB infers that a new value is computed whenever the accessed stream produces a value. (Remember the asynchronous nature of input streams.) For more details on this and especially the case, where you access multiple streams, take a look at the [reference on when output streams compute a new value](#).

StreamLAB also tries to infer the data-type. We could also explicitly annotate the stream just like the input streams:

```
output number_of_satellites_too_large : Bool := gps > 16
```

If `number_of_satellites_too_large` becomes true, we want StreamLAB to raise an alarm. We therefore define a *trigger* with a meaningful message:

```
trigger number_of_satellites_too_large
    "Number of satellites is too large for flying near the ground."
```

We also add triggers for insufficient satellites:

```
trigger gps = 3
    "Few GPS satellites in range. 3 dimensional location unavailable"
trigger gps < 3
    "Few GPS satellites in range. No GPS location available."
```

You can now run StreamLAB on the provided input trace `line.csv` with the specification we wrote so far:

```
// number of visible GPS satellites
input gps: UInt64
output number_of_satellites_too_large := gps > 16
trigger number_of_satellites_too_large
    "Number of satellites is too large for flying near the ground."
trigger gps = 3
    "Few GPS satellites in range. 3 dimensional location unavailable"
trigger gps < 3
    "Few GPS satellites in range. No GPS location available."
```

You should use the `triggers` verbosity and a relative time representation such as `relative_human` so that StreamLAB prints all triggers with their respective time.

```
streamlab monitor spec01.lola --offline --csv-in line.csv --
time-info-rep relative_human
```

You should see messages like:

```
22m 35s 155ms 175us: Trigger: Number of satellites is too large for flyin
22m 44s 726ms 949us: Trigger: Few GPS satellites in range. No GPS locatio
22m 49s 350ms 658us: Trigger: Few GPS satellites in range. 3 dimensional
```

Checking the GPS Module's Frequency

Now we will use StreamLAB's real-time and aggregation capabilities to validate the GPS module's output frequency of 2Hz. We declare two additional input streams:

```
input lat: Float64
input lon: Float64
```

We will now count the number of readings for each of the two streams in a sliding-window over the last four second plus some epsilon. We do not want to make the computation of a new value depend on some other stream producing a new value, but instead compute a new value at a frequency of 20Hz. By adding the `@ 20Hz` tag after the stream name, we mark the stream as **periodic** instead of **event-based**.

```
output not_enough_lat_readings @ 10Hz
  := lat.aggregate(over: 4.1s, using:count) < 7
output not_enough_lon_readings @ 10Hz
  := lon.aggregate(over_exactly: 4.1s, using:count).defaults(to:7) < 7
```

Here we used two different semantics for **sliding-windows**, which only differ in the initial phase of monitoring. While not sufficient time has passed to fill the sliding window, using `aggregate` with `over` already returns a value. Using `over_exactly` starts by returning `None` and the aggregation returns a value only once the complete window size was observed. That is why we have to declare a **default value** to define what value should be used when the aggregation returns `None`. Take a look at the [reference on sliding-windows](#) for more information on restrictions and the available aggregation functions.

We again add a trigger for each of the two streams:

```
trigger not_enough_lat_readings
  "GPS module did not provide sufficient lat readings.(Ignore initially)"
trigger not_enough_lon_readings
  "GPS module did not provide sufficient lon readings."
```

Initially the trigger for the latitude readings will fire several times because the number of observed readings is still too low. The trigger for the longitude readings will not fire, because we used a default for the sliding window that is sufficient to satisfy our check.

Now try to run StreamLAB on the provided input trace `line.csv` with the specification we wrote so far:

```
// number of visible GPS satellites
input gps: UInt64
output number_of_satellites_too_large := gps > 16
trigger number_of_satellites_too_large
```

```

    "Number of satellites is too large for flying near the ground."
trigger gps = 3
    "Few GPS satellites in range. 3 dimensional location unavailable"
trigger gps < 3
    "Few GPS satellites in range. No GPS location available."

input lat: Float64
input lon: Float64
output not_enough_lat_readings @ 10Hz
    := lat.aggregate(over: 4.1s, using:count) < 7
output not_enough_lon_readings @ 10Hz
    := lon.aggregate(over_exactly: 4.1s, using:count).defaults(to:7) < 7
trigger not_enough_lat_readings
    "GPS module did not provide sufficient lat readings.(Ignore initially)"
trigger not_enough_lon_readings
    "GPS module did not provide sufficient lon readings."

```

You should use the `triggers` verbosity and a relative time representation such as `relative_human` so that StreamLAB prints all triggers with their respective time.

```

streamlab monitor spec02.lola --offline --csv-in line.csv --
time-info-rep relative_human

```

Approximating the Ground Speed Through GPS

We can use consecutive latitude and longitude readings to calculate an approximate distance. We need `abs`, `cos` and `sqrt` for the calculation so we have to import the `math` module. The formula uses the Pythagorean theorem but first converts the longitude and latitude differences, measured in degree, according to the length of a meridian or respective equator parallel.

```

import math
constant meridian_length_in_meters: Float64 := 200003930.0
constant equator_length_in_meters: Float64 := 40075014.0
output lat_diff := lat - lat.offset(by:-1).defaults(to:lat)
output lon_diff := lon - lon.offset(by:-1).defaults(to:lon)
output lat_avg := (lat + lat.offset(by:-1).defaults(to:lat))/2.0
output lat_diff_m := meridian_length_in_meters / 180.0 * abs(lat_diff)
output lon_diff_m := equator_length_in_meters / 360.0 * cos(lat_avg)
    * abs(lon_diff)

```

```
output distance_gps := sqrt(lat_diff_m**2.0 + lon_diff_m**2.0)
```

This formula gives a good approximation for small distances. For distances less than 1 mile the error is below 0.5%. We used **constants** to explain the used factors.

The approach above works but only if we receive new latitude and longitude readings at the same time. By using **synchronous access**, meaning just using another stream's name, we tell StreamLAB to compute a new value only if that stream also produces a new value. By accessing `lat` and `lon`, StreamLAB infers the conjunct of these two streams as the **activation condition** for computing a new value. Due to asynchronous input streams, we could have a module provide latitude and longitude readings sequentially. In this case `lon_diff_m` and `distance_gps` would never be computed.

For the scenario that the GPS module first provides longitude and then latitude readings we could fix this by replacing the synchronous access of `lon`. We can use a zero-order **hold** on `lon` and thereby remove the dependency on having a new value at the same point in time.

```
import math
constant meridian_length_in_meters: Float64 := 200003930.0
constant equator_length_in_meters: Float64 := 40075014.0
constant start_lon: Float64 := 0.0
output lon_offset := lon.offset(by:-1).defaults(to:lon)
output lat_diff := lat - lat.offset(by:-1).defaults(to:lat)
output lon_diff @ lat := lon.hold().defaults(to:start_lon)
  - lon_offset.hold().defaults(to:start_lon)
output lat_avg := (lat + lat.offset(by:-1).defaults(to:lat))/2.0
output lat_diff_m := meridian_length_in_meters / 180.0 * abs(lat_diff)
output lon_diff_m := equator_length_in_meters / 360.0 * cos(lat_avg)
  * abs(lon_diff)
output distance_gps := sqrt(lat_diff_m**2.0 + lon_diff_m**2.0)
```

Due to the hold, `lon_diff` will now be evaluated whenever `lat` provides a new value.

Now we will assume that we also have a clock running that gives us the current time whenever any input stream produces a value. For offline

analysis we need such time-stamps anyway. In our case the clock will be providing a `time` stream based on seconds. We use the computed distance and the time difference between `lat` readings.

```
input time : Float64
output lat_timestamp @ lat := time.hold().defaults(to:0.0)
output time_difference := lat_timestamp
  - lat_timestamp.offset(by:-1).defaults(to:lat_timestamp-0.1)
output speed := distance_gps / time_difference
```

Another option would be to sample the latitude and longitude readings with a fixed frequency and determine the speed based on the position 1 second ago.

```
import math
output sampled_lat @ 10Hz := lat.hold().defaults(to:0.0)
output sampled_lon @ 10Hz := lon.hold().defaults(to:0.0)
output sampled_lon_diff := sampled_lon
  - sampled_lon.offset(by:-1s).defaults(to:sampled_lon)
output sampled_lat_diff := sampled_lat
  - sampled_lat.offset(by:-1s).defaults(to:sampled_lat)
output sampled_lat_avg := (sampled_lat + sampled_lat.offset(by:-1s)
  .defaults(to:sampled_lat))/2.0
output sampled_lat_diff_m := meridian_length_in_meters / 180.0
  * abs(sampled_lat_diff)
output sampled_lon_diff_m := equator_length_in_meters / 360.0
  * cos(sampled_lat_avg) * abs(sampled_lon_diff)
output sampled_speed := sqrt( sampled_lat_diff_m**2.0
  + sampled_lon_diff_m**2.0)
```

Now try to run StreamLAB on the provided input trace `line.csv` with the specification we wrote:

```
import math
constant meridian_length_in_meters: Float64 := 200003930.0
constant equator_length_in_meters: Float64 := 40075014.0
constant start_lon: Float64 := 0.0
input lat: Float64
input lon: Float64
```

```

output sampled_lat @ 10Hz := lat.hold().defaults(to:0.0)
output sampled_lon @ 10Hz := lon.hold().defaults(to:0.0)
output sampled_lon_diff := sampled_lon
    - sampled_lon.offset(by:-1s).defaults(to:sampled_lon)
output sampled_lat_diff := sampled_lat
    - sampled_lat.offset(by:-1s).defaults(to:sampled_lat)
output sampled_lat_avg := (sampled_lat + sampled_lat.offset(by:-1s)
    .defaults(to:sampled_lat))/2.0
output sampled_lat_diff_m := meridian_length_in_meters / 180.0
    * abs(sampled_lat_diff)
output sampled_lon_diff_m := equator_length_in_meters / 360.0
    * cos(sampled_lat_avg) * abs(sampled_lon_diff)
output sampled_speed := sqrt( sampled_lat_diff_m**2.0
    + sampled_lon_diff_m**2.0)

```

You should use the `triggers` verbosity and a relative time representation such as `relative_human` so that StreamLAB prints all triggers with their respective time.

```

streamlab monitor spec03.lola --offline --csv-in line.csv --
time-info-rep relative_human

```

Provided Tool

The `streamlab` has one subcommand for analyzing a specification and one for monitoring. The input files should be UTF-8 encoded without a byte order mark (BOM).

streamlab monitor

`streamlab` provides an `--online` and an `--offline` monitoring mode.

- In the online mode, StreamLAB can read CSV based input from STDIN (`--stdin`).
- In the offline mode, a CSV file has to be used (`--csv-in`). In this case one of the columns must contain the timestamps.

The output is determined by the `--verbosity` and `--time-info-rep` flags. Where the output is sent can be controlled with `--stdout` and `--stderr`. Take a look at the help provided by `streamlab help monitor` for more detailed information.

Example invocations:

- `streamlab monitor --offline --csv-in ./PATH/TO/CSV --verbosity triggers --time-info-rep relative ./PATH/TO/SPEC`
- `streamlab monitor --online --stdin --verbosity progress ./PATH/TO/SPEC`

streamlab analyze

`streamlab analyze` can be used for type-checking and static analysis of a specification.

Consult the built-in help for more information. In the initial version

`streamlab analyze` only checks for errors.

Example invocations:

- `streamlab analyze ./PATH/TO/SPEC`

RTLola Reference

Comments

RTLola allows two common types of comments:

```
// comments until the end of the line

/*
Comments between the starting and ending delimiter.
The delimiters do not nest.
*/
```

Available Data Types

- `UInt64`
- `Int64`
- `Float64`
- `Bool`
- `String`

Defining Input Streams

Input streams are declared with the `input` keyword followed by an identifier, a colon, and a type name.

```
input STREAM_NAME : TYPE_NAME
```

Not all input streams have to produce a value at the same time.

Defining Output Streams

Every output stream has a data-type and a **stream type**. There are two stream types of output streams:

- One type is called **periodic** and produces new values at a fixed rate.
- The other type is called **event-based** and produces a new value whenever the required input streams produce a new value.

If the stream type is not specified explicitly, StreamLAB tries to infer it from the stream's expression. The same holds for the data-type of the stream.

Event-Based Output Streams

```
output STREAM_NAME : TYPE_NAME @ ACTIVATION_CONDITION := EXPRESSION
output STREAM_NAME : TYPE_NAME                        := EXPRESSION
output STREAM_NAME @ ACTIVATION_CONDITION := EXPRESSION
output STREAM_NAME                                     := EXPRESSION
```

When do They Compute a New Value?

Event-based streams compute a new value whenever an input event occurs and the **activation condition** holds. It is a boolean expression over the stream names using the boolean connectives \wedge and \vee .

The inferred activation condition is the conjunction of all streams that are synchronously accessed in the stream's expression, meaning that all these streams produce a new value at this exact moment. The activation condition can further be restricted by explicitly stating it in the stream definition. The explicit activation condition must imply the inferred one to guarantee new values for the synchronous accesses.

Periodic Output Streams

```
output STREAM_NAME : TYPE_NAME @ FREQUENCY := EXPRESSION
output STREAM_NAME : TYPE_NAME := EXPRESSION
output STREAM_NAME @ FREQUENCY := EXPRESSION
output STREAM_NAME := EXPRESSION
```

Periodic output streams produce a value at a fixed rate.

Accessing a Stream

Synchronous Access

STREAM_NAME

A synchronous access is expressed by using a stream's name. It signifies the named stream's value that was produced at the same point in time. To guarantee that the named stream produced a value at the same time, the type system enforces some constraints:

For synchronous access expressions in event-based streams, the accessed stream must also be event-based and implied by the activation condition.

For synchronous access expressions in periodic streams, the accessed stream must also be periodic with a frequency that is a multiple of the accessing stream.

Offset Based Access

STREAM_NAME.offset (by: OFFSET) .defaults (to:DEFAULT_EXPRESSION)

Offset based access is a special form of synchronous look up. An integer based offset can be used to use a different value other than the current one. Negative offsets access past values. A time-based offset can only access periodic streams. In this case, the offset has to be a multiple of the period of the accessed stream. Due to the possibility that the accessed stream did not produce the requested value, the value of the offset based access needs to define a default.

Optional Access

STREAM_NAME.get () .defaults (to:DEFAULT_EXPRESSION)

An optional access is expressed by using a stream's name and calling the `.get()` method on it. It signifies the named stream's value that was produced at the same point in time, if it exists. Due to the possibility that the accessed stream did not produce a value, the value of the optional access needs to define a default. The accessed stream must have the same stream type as the accessing one.

Zero-Order Hold

```
STREAM_NAME.hold().defaults(to:DEFAULT_EXPRESSION)
```

A zero-order hold access is expressed by using a stream's name and calling the `.hold()` method on it. It signifies the named stream's last value that was produced up through this point in time, if it exists. Due to the possibility that the accessed stream did not produce a value, the value of the hold access needs to define a default.

Sliding-Windows

```
STREAM_NAME.aggregate(over:          DURATION, using: AGGREGATION)
STREAM_NAME.aggregate(over:          DURATION, using: AGGREGATION)
               .defaults(to: DEFAULT_EXPRESSION)
STREAM_NAME.aggregate(over_exactly: DURATION, using: AGGREGATION)
               .defaults(to: DEFAULT_EXPRESSION)
```

A sliding-window aggregates the named stream's values that were produced in a time-window from now into the past with a size specified by `DURATION`. There are two semantics that differ only initially while the monitor has run for a time less than the specified `DURATION`:

- Using `over: DURATION` causes the aggregation to produce a value even when the monitor did not observe the full duration.
- Using `over_exactly: DURATION` would produce `None` while the full duration was not observed. Such an aggregation therefore needs to define a default.

If the aggregation function does not have a meaningful behavior for special cases, such as no or only one observed event, a default must be specified. Sliding-windows are only allowed in the expressions of periodic streams.

Available Aggregations

- avg
- count
- integral
- max
- min
- sum