Tiny Drone, Tiny Resources: Deploying RTLola Monitors on the Crazyflie Platform

Jan Baumeister[1], Vihaan Bhaduri[2], Bernd Finkbeiner[1], Florian Kohn[1], and Frederik Scheerer[1]

[1] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
`{jan.baumeister, finkbeiner, florian.kohn, frederik.scheerer}@cispa.de`
[2] Saratoga High School, Saratoga, USA
`vihaan.bhaduri@gmail.com`

Abstract. The Crazyflie is an open-source micro quadcopter weighing 29g that can be operated indoors without special safety requirements. This characteristic makes it a practical platform for developing and testing runtime monitors beyond simulation. Its lightweight design also makes it prone to disturbances such as drift, similar to larger drones in outdoor environments, thereby providing realistic data without the need for specially prepared environments. The Crazyflie is based on an STM32 microcontroller with 92kb of SRAM and 1Mb of flash memory shared between the flight controller and the monitor. In this paper, we present the integration of the stream-based monitoring framework RTLola on the Crazyflie. We describe how bounded memory monitors can be generated for embedded devices, analyzing the trade-off between specification size and the use of complex data structures. The evaluation is conducted on a waypoint mission, tracking the absolute position of the drone with millimeter precision.

Keywords: Stream-based Runtime Monitoring · Crazyflie · Embedded Systems.

1 Introduction

The Crazyflie [1] is an open-source micro-drone platform based on the STM32 microcontroller. It includes an internal inertial measurement unit and flight controller and can be extended with *decks*, such as the *Multi-Ranger-Deck*, which measures distances to obstacles in five directions using time-of-flight sensors. Due to its small size and weight, it can be operated without a license or strict safety precautions. At the same time, its low weight makes it susceptible to disturbances such as drift in indoor flight, comparable to those experienced by larger drones outdoors, which makes it a suitable platform for developing and testing runtime monitors. Images of the drone used for the experiments presented in this case study are shown in Figure 1

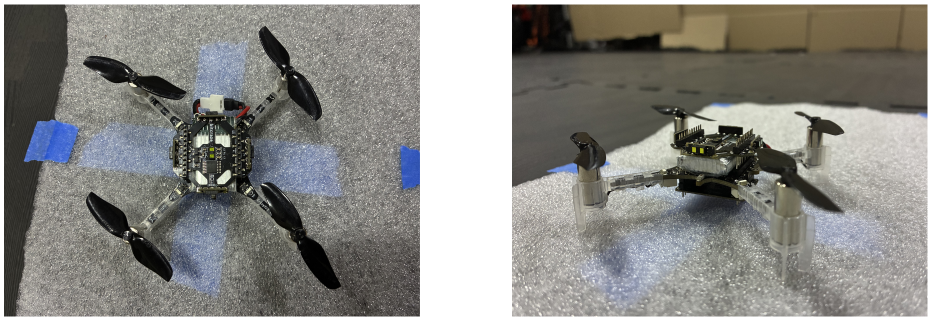

(a) The top view of the drone. (b) The side view of the drone.

Fig. 1: The Crazyflie drone used for the presented experiments.

RTLola [3,4] is a stream-based runtime monitoring framework that has previously been applied to larger drones capable of carrying payloads of up to 200kg. Prior work demonstrated that monitor integration must be tested and verified throughout all development stages of unmanned aircraft, including hardware-in-the-loop testing prior to flight tests. In these case studies [3,6,7], the systems under observation could support dedicated monitoring hardware due to more relaxed payload restrictions. By contrast, the Crazyflie has a maximal recommended payload of 15g, which imposes strict limitations on additional hardware.

This case study investigates how a runtime monitor can be integrated directly into the Crazyflie's onboard embedded hardware, eliminating the need for additional monitoring components. Using the RTLola Compiler [2] to generate C code, we evaluate the binary size and memory requirements of monitors derived from a comprehensive set of specifications. We compare two approaches: one using parameterized streams, which require complex data structures to manage multiple stream instances, and another in which parameterized streams are unrolled prior to compilation, resulting in significantly larger specification sizes.

2 The Crazyflie Platform

The Crazyflie project provides an open-source software stack supporting both ground-based and onboard control and logging. Its internal motion commander regulates the drone's movement. The primary development entry point is the *cflib*[3] a Python library enabling communication with the drone from the ground via a USB-powered transmitter. The drone firmware is also fully open source and can be modified, for instance, by implementing applications through the

[3] `https://www.bitcraze.io/documentation/repository/crazyflie-lib-python/master/`

application API[4] In the following, we summarize the properties monitored during the experiments through these API's.

2.1 Position Inputs

For the experiments, two types of position data were monitored. First, the *Multi-Ranger* and *Z-Ranger* decks measured distances to the flight cage boundaries using time-of-flight (ToF) sensors. Second, position estimates were obtained from the *stateEstimate* logging group.

– `range.front`: Distance from the front sensor to an obstacle [mm].
– `range.back`: Distance from the back sensor to an obstacle [mm].
– `range.up`: Distance from the top sensor to an obstacle [mm].
– `range.left`: Distance from the left sensor to an obstacle [mm].
– `range.right`: Distance from the right sensor to an obstacle [mm].
– `range.zrange`: Distance from the Z-ranger (bottom) sensor to an obstacle [mm].
– `stateEstimate.x`: Estimated position in the global reference frame, X [m].
– `stateEstimate.y`: Estimated position in the global reference frame, Y [m].
– `stateEstimate.z`: Estimated position in the global reference frame, Z [m].

3 RTLola

In this section, we provide an overview of RTLola using the waypoint specification applied in Section 5. The specification defines a monitor that verifies whether the drone passes a sequence of waypoints in the prescribed order. Any deviation from the path to the next waypoint is classified as a violation. For simplicity, we restrict the setting to two dimensions and assume that the waypoints are provided to the monitor prior to the start of the drone.

```
import math
input x : Float32
input y : Float32

constant UNREACHABLE: (Float64, Float64) := (-1.0, -1.0)

input wp_x: Float32
input wp_y: Float32
output waypoint_idx:UInt @wp_x :=
       waypoint_idx.offset(by: -1).defaults(to: 0) + 1

output waypoint(idx: UInt)
  spawn with waypoint_idx
  eval when waypoint_idx == idx with (wp_x, wp_y)

```

[4] https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/userguides/app_layer/

```
16 output distance_to_waypoint(idx: UInt)
17   spawn with waypoint_idx
18   eval with sqrt((x - waypoint(idx).hold(or: UNREACHABLE).0)**2.0 +
         (y - waypoint(idx).hold(or: UNREACHABLE).1)**2.0)
19
20 output waypoint_reached(idx: UInt)
21   spawn with waypoint_idx
22   eval with waypoint_reached(idx).offset(by: -1).defaults(to: false)
23                 || (current_waypoint == idx &&
                       distance_to_waypoint(idx) < 25.0)
24
25 output unreached_waypoints(idx: UInt)
26   spawn with waypoint_idx
27   eval @(x&&y) with idx
28   close @(x&&y) when waypoint_reached(idx).get(or:false)
29
30 output current_waypoint @(x&&y) :=
31     unreached_waypoints
32         .aggregate(over_instances:all, using: min).defaults(to: 0)
33
34 output finished @(x&&y) :=
      waypoint_reached.aggregate(over_instances: all, using: forall)
35 trigger finished "All Waypoints reached successfully"
36
37 output distance_to_next @(x&&y) :=
      distance_to_waypoint(current_waypoint).hold(or: 999.0)
38 output distance_to_next_increased := distance_to_next >
      distance_to_next.offset(by: -1).defaults(to: 999.0)
39 trigger @2Hz
40     distance_to_next_increased.aggregate(over: 5s, using: forall)
41     "Drifting away from next waypoint"
```

Listing 1.1: An RTLola specification used to monitor that the Crazyflie passes a series of waypoints in order.

The input streams `wp_x` and `wp_y` capture the coordinates of the waypoints provided to the monitor. The drone is expected to pass these waypoints in the given order. The `waypoint_idx` stream assigns an index to each waypoint, and these streams are combined into the parameterized output stream `waypoint`.

A *parameterized output stream* in RTLola represents a set of stream instances, each identified by a unique parameter combination. Stream instances are created and removed dynamically at runtime. The `waypoint` stream is parameterized by a single parameter *idx*. Its `spawn` clause (line 13) specifies that a new stream instance is created for each waypoint, with the parameter value assigned from the current value of `waypoint_idx`. The `eval` clause defines that each stream instance evaluates to the coordinates of the corresponding waypoint. The condition following the `when` keyword ensures that evaluation occurs once, when the waypoint is received.

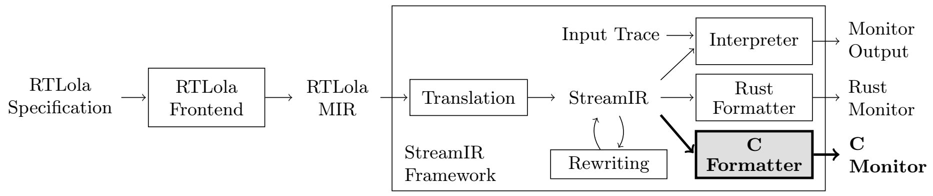


Fig. 2: Overview of the StreamIR framework

The following streams are parameterized in the same way, ensuring that computations are performed separately for each waypoint. The `distance_to_waypoint` stream computes the distance between the drone's coordinates (given as the input streams `x`, `y`) and each waypoint. For instance, the stream instance with parameter 0 of `distance_to_waypoint` represents the distance to the first waypoint. The `waypoint_reached` stream determines whether the drone approached a waypoint up to a predefined threshold. The `unreached_waypoints` stream collects the indices of waypoints not yet reached. This behavior is implemented with the `close` clause, which removes an instance once the corresponding waypoint has been reached. From this, the `current_waypoint` stream is derived by selecting the minimal index among the unreached waypoints.

Finally, two trigger conditions are defined to notify the operator. The first trigger (line 35) signals when all waypoints have been reached. The second trigger (line 39) signals when the distance to the next waypoint has increased continuously for five seconds.

4 Extending the RTLola Compiler

The StreamIR framework [2] introduces an intermediate representation, StreamIR, which serves as a compilation step for translating RTLola into multiple target languages. While RTLola specifications define streams relationally through stream expressions, StreamIR provides an imperative representation of the monitor. In contrast to general-purpose intermediate representations such as LLVM, StreamIR remains closely aligned with stream-based languages, offering constructs such as explicit stream evaluation. This design enables optimizations beyond those possible at the specification level [5]. Furthermore, because StreamIR preserves assumptions specific to stream-based semantics, it permits additional optimizations that cannot be performed by the compiler of the target language, as shown in [2]. Building on this framework, for this project we extend StreamIR with a backend for compilation to C, enabling efficient use of RTLola on a microcontroller.

Figure 2 provides an overview of the StreamIR framework. First, the RTLola specification is parsed by the RTLola frontend, which produces the RTLolaMIR, a representation enriched with analysis results but still closely resembling the original specification. This MIR is then translated into StreamIR, which supports a wide range of optimizations through rewriting rules applied to its structure. After optimization, the StreamIR can either be interpreted via just-in-time compilation or compiled into different target languages. For the purpose of this work, we extended the framework with a backend that generates highly memory-efficient C code suitable for execution on microcontrollers.

A critical requirement for running monitors on microcontrollers is a bounded, heapless memory footprint. Some specifications used in our drone applications rely on parameterized streams, which by default may require unbounded memory. To address this, we introduced annotations for parameterized streams that allow the user to specify an upper bound on the number of stream instances. Eviction strategies then ensure that the number of active instances never exceeds this bound, restoring memory boundedness. This mechanism enables the generation of heapless code, which is essential for executing monitors on microcontrollers without relying on a dynamic memory allocator.

5 Implementation & Evaluation

For the evaluation, we compile RTLola specifications to C and report the memory usage of the produced code and the binary size. Since the drone has limited hardware capabilities, the monitor needs to be efficient not only regarding runtime but also regarding memory.

We have evaluated our approach on three specifications, monitoring the position of the drone. The first specification is described in Section 3 and checks if, given a set of waypoints, all waypoints are reached during the flight. These waypoints describe the mission of the flight test. The second specification describes a tube around the flight path of the drone to check if it diverges from the plan. This specification is used to detect early errors in the flight controller. The last specification describes a geofence from the drone, i.e., an area where the drone can fly. If the drone violates this specification, a countermeasure is initiated to land the drone immediately.

We report the results of the experiments in Table 1. It shows the compiled size of the different specifications for the Crazyflie microprocessor, separated into Flash and RAM usage. Furthermore, we differentiate between the bounded parameterized implementation and an unrolled variant, where each parameterized stream instance is listed in the specification as a separate stream definition.

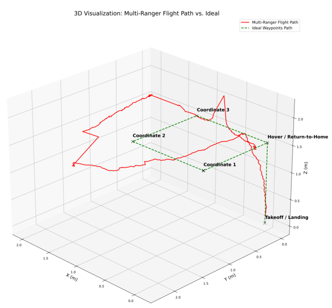
As expected, the unrolled variants are predominantly larger than their parameterized counterparts in both Flash and RAM. The reduction in code size is particularly pronounced for geofence and waypoints, where parameterized implementations drastically reduce LOC and Flash. Tube is an exception: its RAM usage is higher for the parameterized version, but the reduction in Flash clearly dominates.

To test the specifications, we integrated the generated monitors into the Crazyflie setup and executed an autonomous waypoint mission. Figure 3a shows the flight area on the left. All flights were conducted in a net-enclosed environment to ensure the safety of both the drone and the operators. Figure 3 illustrates a representative flight. The red line indicates the actual trajectory of the drone,

Spec	**Parameterized**			**Unrolled**		
	LOC	Flash	RAM	LOC	Flash	RAM
geofence	92	32.1 KB	0.6 KB	483	50.8 KB	1.0 KB
tube	111	33.6 KB	3.6 KB	344	47.0 KB	1.2 KB
waypoints	37	13.8 KB	0.3 KB	100	21.2 KB	0.4 KB

Table 1: Comparison of LOC, Flash (text+data) and RAM (bss+data) for parameterized and unrolled variants

(a) The view of the flight area.



(b) A flight of the drone (red line), the waypoints (black crosses), and the intended flight path (green line)

Fig. 3: The Crazyflie drone used for the presented experiments.

the green line represents the planned trajectory, and the black crosses mark the waypoints as described in Section 3. The observed deviation from the intended path highlights the importance of monitoring during flight.

6 Conclusion

In this case study, we examined the integration of runtime monitors into the Crazyflie micro-drone. To this end, we extended the RTLola compiler framework with a C backend supporting bounded parameterization. Previously, parameterized streams had to be unrolled to generate monitors executable on heapless embedded devices. The bounded parameterization approach allows specifications to remain compact and manageable while maintaining executability on such targets.

We evaluated this approach by comparing the flash and RAM requirements of monitors generated from parameterized and unrolled specifications. The results indicate that monitors based on bounded parameterized specifications require less flash and RAM than those based on unrolled specifications, demonstrating their suitability for embedded runtime monitoring.

A The Waypoint Specification

A.1 The parameterized Specification

```
import math
input x : Float32
input y : Float32

constant UNREACHABLE: (Float32, Float32) := (-1.0, -1.0)

input wp_x: Float32
input wp_y: Float32
output waypoint_idx:UInt @wp_x := waypoint_idx.offset(by:
    -1).defaults(to: 0) + 1

output waypoint(idx: UInt)
  spawn with waypoint_idx
  eval when waypoint_idx == idx with (wp_x, wp_y)

output distance_to_waypoint(idx: UInt)
  spawn with waypoint_idx
  eval with sqrt((x - waypoint(idx).hold(or: UNREACHABLE).0)**2.0 +
      (y - waypoint(idx).hold(or: UNREACHABLE).1)**2.0)

output waypoint_reached(idx: UInt)
  spawn with waypoint_idx
  eval with waypoint_reached(idx).offset(by: -1).defaults(to: false)
                || (current_waypoint == idx &&
                    distance_to_waypoint(idx) < 25.0)

output unreached_waypoints(idx: UInt)
  spawn with waypoint_idx
  eval @(x&&y) with idx
  close @(x&&y) when waypoint_reached(idx).get(or:false)

output current_waypoint @(x&&y) :=
    unreached_waypoints.aggregate(over_instances:all, using:
    min).defaults(to: 0)

output finished @(x&&y) :=
    waypoint_reached.aggregate(over_instances: all, using: forall)
trigger finished "All Waypoints reached successfully"

output distance_to_next @(x&&y) :=
    distance_to_waypoint(current_waypoint).hold(or: 999.0)
output distance_to_next_increased := distance_to_next >
    distance_to_next.offset(by: -1).defaults(to: 999.0)
trigger @2Hz distance_to_next_increased.aggregate(over: 5s, using:
    forall) "Drifting away from next waypoint"
```

A.2 The unrolled Specification

```
import math
input x : Float32
input y : Float32

constant UNREACHABLE: (Float32, Float32) := (-1.0, -1.0)

constant waypoint0 : (Float32, Float32) := (0.0, 0.0)
constant waypoint1 : (Float32, Float32) := (2.0, 0.0)
constant waypoint2 : (Float32, Float32) := (2.0, 1.0)
constant waypoint3 : (Float32, Float32) := (2.0, 2.0)
constant waypoint4 : (Float32, Float32) := (3.0, 2.0)
constant waypoint5 : (Float32, Float32) := (2.0, 3.0)
constant waypoint6 : (Float32, Float32) := (5.0, 3.0)
constant waypoint7 : (Float32, Float32) := (3.0, 3.0)
constant waypoint8 : (Float32, Float32) := (2.0, 1.0)
constant waypoint9 : (Float32, Float32) := (0.0, 0.0)

output distance_to_waypoint0
  eval with sqrt((x - waypoint0.0)**2.0 + (y - waypoint0.1)**2.0)
output distance_to_waypoint1
  eval with sqrt((x - waypoint1.0)**2.0 + (y - waypoint1.1)**2.0)
output distance_to_waypoint2
  eval with sqrt((x - waypoint2.0)**2.0 + (y - waypoint2.1)**2.0)
output distance_to_waypoint3
  eval with sqrt((x - waypoint3.0)**2.0 + (y - waypoint3.1)**2.0)
output distance_to_waypoint4
  eval with sqrt((x - waypoint4.0)**2.0 + (y - waypoint4.1)**2.0)
output distance_to_waypoint5
  eval with sqrt((x - waypoint5.0)**2.0 + (y - waypoint5.1)**2.0)
output distance_to_waypoint6
  eval with sqrt((x - waypoint6.0)**2.0 + (y - waypoint6.1)**2.0)
output distance_to_waypoint7
  eval with sqrt((x - waypoint7.0)**2.0 + (y - waypoint7.1)**2.0)
output distance_to_waypoint8
  eval with sqrt((x - waypoint8.0)**2.0 + (y - waypoint8.1)**2.0)
output distance_to_waypoint9
  eval with sqrt((x - waypoint9.0)**2.0 + (y - waypoint9.1)**2.0)

output waypoint0_reached
  eval @(x&&y) with waypoint0_reached.offset(by:-1).defaults(to:
      false) || (current_waypoint.hold(or: 0) == 0 &&
      distance_to_waypoint0.hold(or: 0.0) < 25.0)
output waypoint1_reached
  eval @(x&&y) with waypoint1_reached.offset(by:-1).defaults(to:
      false) || (current_waypoint.hold(or: 0) == 1 &&
      distance_to_waypoint1.hold(or: 0.0) < 25.0)
output waypoint2_reached
```

```
  eval @(x&&y) with waypoint2_reached.offset(by:-1).defaults(to:
      false) || (current_waypoint.hold(or: 0) == 2 &&
      distance_to_waypoint2.hold(or: 0.0) < 25.0)
output waypoint3_reached
  eval @(x&&y) with waypoint3_reached.offset(by:-1).defaults(to:
      false) || (current_waypoint.hold(or: 0) == 3 &&
      distance_to_waypoint3.hold(or: 0.0) < 25.0)
output waypoint4_reached
  eval @(x&&y) with waypoint4_reached.offset(by:-1).defaults(to:
      false) || (current_waypoint.hold(or: 0) == 4 &&
      distance_to_waypoint4.hold(or: 0.0) < 25.0)
output waypoint5_reached
  eval @(x&&y) with waypoint5_reached.offset(by:-1).defaults(to:
      false) || (current_waypoint.hold(or: 0) == 5 &&
      distance_to_waypoint5.hold(or: 0.0) < 25.0)
output waypoint6_reached
  eval @(x&&y) with waypoint6_reached.offset(by:-1).defaults(to:
      false) || (current_waypoint.hold(or: 0) == 6 &&
      distance_to_waypoint6.hold(or: 0.0) < 25.0)
output waypoint7_reached
  eval @(x&&y) with waypoint7_reached.offset(by:-1).defaults(to:
      false) || (current_waypoint.hold(or: 0) == 7 &&
      distance_to_waypoint7.hold(or: 0.0) < 25.0)
output waypoint8_reached
  eval @(x&&y) with waypoint8_reached.offset(by:-1).defaults(to:
      false) || (current_waypoint.hold(or: 0) == 8 &&
      distance_to_waypoint8.hold(or: 0.0) < 25.0)
output waypoint9_reached
  eval @(x&&y) with waypoint9_reached.offset(by:-1).defaults(to:
      false) || (current_waypoint.hold(or: 0) == 9 &&
      distance_to_waypoint9.hold(or: 0.0) < 25.0)

output finished @(x&&y) := waypoint0_reached &&
  waypoint1_reached &&
  waypoint2_reached &&
  waypoint3_reached &&
  waypoint4_reached &&
  waypoint5_reached &&
  waypoint6_reached &&
  waypoint7_reached &&
  waypoint8_reached &&
  waypoint9_reached

trigger finished "All Waypoints reached successfully"

output current_waypoint
  eval @(x&&y) when !waypoint0_reached.offset(by:-1).defaults(to:
      false) with 0
  eval @(x&&y) when !waypoint1_reached.offset(by:-1).defaults(to:
      false) with 1
  eval @(x&&y) when !waypoint2_reached.offset(by:-1).defaults(to:
      false) with 2
  eval @(x&&y) when !waypoint3_reached.offset(by:-1).defaults(to:
      false) with 3
  eval @(x&&y) when !waypoint4_reached.offset(by:-1).defaults(to:
      false) with 4
  eval @(x&&y) when !waypoint5_reached.offset(by:-1).defaults(to:
      false) with 5
  eval @(x&&y) when !waypoint6_reached.offset(by:-1).defaults(to:
      false) with 6
  eval @(x&&y) when !waypoint7_reached.offset(by:-1).defaults(to:
      false) with 7
  eval @(x&&y) when !waypoint8_reached.offset(by:-1).defaults(to:
      false) with 8
  eval @(x&&y) when !waypoint9_reached.offset(by:-1).defaults(to:
      false) with 9

output distance_to_next
  eval @(x&&y) when current_waypoint.hold(or: 0) == 0 with
      distance_to_waypoint0
  eval @(x&&y) when current_waypoint.hold(or: 0) == 1 with
      distance_to_waypoint1
  eval @(x&&y) when current_waypoint.hold(or: 0) == 2 with
      distance_to_waypoint2
  eval @(x&&y) when current_waypoint.hold(or: 0) == 3 with
      distance_to_waypoint3
  eval @(x&&y) when current_waypoint.hold(or: 0) == 4 with
      distance_to_waypoint4
  eval @(x&&y) when current_waypoint.hold(or: 0) == 5 with
      distance_to_waypoint5
  eval @(x&&y) when current_waypoint.hold(or: 0) == 6 with
      distance_to_waypoint6
  eval @(x&&y) when current_waypoint.hold(or: 0) == 7 with
      distance_to_waypoint7
  eval @(x&&y) when current_waypoint.hold(or: 0) == 8 with
      distance_to_waypoint8
  eval @(x&&y) when current_waypoint.hold(or: 0) == 9 with
      distance_to_waypoint9

output distance_to_next1 @(x&&y) := distance_to_next.hold(or: 0.0)

output distance_to_next_increased @(x&&y) := distance_to_next1 >
    distance_to_next1.offset(by: -1).defaults(to: 999.0)
trigger @2Hz distance_to_next_increased.aggregate(over: 5s, using:
    forall) "Drifting away from next waypoint"
```

B The Tube Specification

B.1 The parameterized Specification

```
import math
input x : Float32
input y : Float32
input z : Float32
input heading : Float32
input velocity : Float32

// Tube Points
input p0_x: Float32
input p0_y: Float32
input p0_z: Float32
input p1_x: Float32
input p1_y: Float32
input p1_z: Float32

constant threshold: Float32 := 350.0

// Knots to m/s
output velocity_ms := velocity * 0.514444

constant PI : Float32 := 3.14519
constant c_epsilon : Float32 := 0.0000001

// Direction vector of line segment
#[bounded="10"]
output ld(x0, y0, z0, x1, y1, z1)
    spawn with (p0_x, p0_y, p0_z, p1_x, p1_y, p1_z)
    eval @x with (x1 - x0, y1 - y0, z1 - z0)

// Direction vector from start of line to current position
#[bounded="10"]
output lc(x0, y0, z0, x1, y1, z1)
    spawn with (p0_x, p0_y, p0_z, p1_x, p1_y, p1_z)
    eval with (x - x0, y - y0, z - z0)

// Compute vector projection of flight vector onto direction vector
     of line
#[bounded="10"]
output lfp_len(x0, y0, z0, x1, y1, z1)
    spawn with (p0_x, p0_y, p0_z, p1_x, p1_y, p1_z)
    eval with (ld(x0, y0, z0, x1, y1, z1).0 * lc(x0, y0, z0, x1, y1,
         z1).0 + ld(x0, y0, z0, x1, y1, z1).1 * lc(x0, y0, z0, x1,
         y1, z1).1 + ld(x0, y0, z0, x1, y1, z1).2 * lc(x0, y0, z0,
         x1, y1, z1).2) / (ld(x0, y0, z0, x1, y1, z1).0 ** 2.0 +
         ld(x0, y0, z0, x1, y1, z1).1**2.0 + ld(x0, y0, z0, x1, y1,
         z1).2**2.0)
```

```

#[bounded="10"]
output lfp_proj(x0, y0, z0, x1, y1, z1)
   spawn with (p0_x, p0_y, p0_z, p1_x, p1_y, p1_z)
   eval with (lfp_len(x0, y0, z0, x1, y1, z1) * ld(x0, y0, z0, x1,
       y1, z1).0, lfp_len(x0, y0, z0, x1, y1, z1) * ld(x0, y0, z0,
       x1, y1, z1).1, lfp_len(x0, y0, z0, x1, y1, z1) * ld(x0, y0,
       z0, x1, y1, z1).2)

#[bounded="10"]
output lfp(x0, y0, z0, x1, y1, z1)
   spawn with (p0_x, p0_y, p0_z, p1_x, p1_y, p1_z)
   eval with (x0 + lfp_proj(x0, y0, z0, x1, y1, z1).0, y0 +
       lfp_proj(x0, y0, z0, x1, y1, z1).1, z0 + lfp_proj(x0, y0,
       z0, x1, y1, z1).2)

#[bounded="10"]
output d_lfp(x0, y0, z0, x1, y1, z1)
   spawn with (p0_x, p0_y, p0_z, p1_x, p1_y, p1_z)
   eval with sqrt((lfp(x0, y0, z0, x1, y1, z1).0 - x)**2.0 +
       (lfp(x0, y0, z0, x1, y1, z1).1 - y)**2.0 + (lfp(x0, y0, z0,
       x1, y1, z1).2 - z)**2.0)

#[bounded="10"]
output d_start(x0, y0, z0, x1, y1, z1)
   spawn with (p0_x, p0_y, p0_z, p1_x, p1_y, p1_z)
   eval with sqrt((x0 - x)**2.0 + (y0 - y)**2.0 + (z0 - z)**2.0)

#[bounded="10"]
output d_end(x0, y0, z0, x1, y1, z1)
   spawn with (p0_x, p0_y, p0_z, p1_x, p1_y, p1_z)
   eval with sqrt((x1 - x)**2.0 + (y1 - y)**2.0 + (z1 - z)**2.0)

#[bounded="10"]
output d_line(x0, y0, z0, x1, y1, z1)
   spawn with (p0_x, p0_y, p0_z, p1_x, p1_y, p1_z)
   eval with if lfp_len(x0, y0, z0, x1, y1, z1) <= 0.0 then
       d_start(x0, y0, z0, x1, y1, z1)
            else if lfp_len(x0, y0, z0, x1, y1, z1) >= 1.0 then
                d_end(x0, y0, z0, x1, y1, z1)
            else d_lfp(x0, y0, z0, x1, y1, z1)

output mdp := d_line.aggregate(over_instances: fresh, using:
    argmin).defaults(to: (0.0,0.0,0.0,0.0,0.0,0.0))

output closest_point_cart := if lfp_len(mdp.0, mdp.1, mdp.2, mdp.3,
    mdp.4, mdp.5).hold(or:0.0) <= 0.0 then (mdp.0, mdp.1, mdp.2)
                          else if lfp_len(mdp.0, mdp.1, mdp.2,
                              mdp.3, mdp.4, mdp.5).hold(or:0.0) >=
                              1.0 then (mdp.3, mdp.4, mdp.5)
                          else lfp(mdp.0, mdp.1, mdp.2, mdp.3,
                              mdp.4, mdp.5).hold(or: (0.0,0.0,0.0))

output closest_point := (closest_point_cart.1, closest_point_cart.0,
    closest_point_cart.2)

// Compute visualization vector
output distance_dir_global := (x - closest_point_cart.0, y -
    closest_point_cart.1, z - closest_point_cart.2)

// Normal vector of the plane
output plane_normal := ld(mdp.0, mdp.1, mdp.2, mdp.3, mdp.4,
    mdp.5).hold(or: (0.0,0.0,0.0))
output plane_normal_len := sqrt(plane_normal.0 ** 2.0 +
    plane_normal.1 ** 2.0 + plane_normal.2 ** 2.0 )
output n := (plane_normal.0/plane_normal_len,
    plane_normal.1/plane_normal_len, plane_normal.2/plane_normal_len)

// Global Up Vector defined as unit vector pointing in z direction,
    or y direction if normal points exactly up
output up: (Float32,Float32,Float32) := if abs(n.2 - 1.0) < 0.000001
    then (0.0,1.0,0.0) else (0.0,0.0,1.0)

// Compute Right vector orthogonal to normal vector and up vector
    using cross product
output right := (n.1 * up.2 - n.2 * up.1, n.2 * up.0 - n.0 * up.2,
    n.0 * up.1 - n.1 * up.0)

// Compute local Up vector orthogonal to the normal vector and the
    right vector using cross product
output lup := (n.1 * right.2 - n.2 * right.1, n.2 * right.0 - n.0 *
    right.2, n.0 * right.1 - n.1 * right.0)

output scale_a := distance_dir_global.0 * right.0 +
    distance_dir_global.1 * right.1 + distance_dir_global.2 * right.2
output scale_b := distance_dir_global.0 * lup.0 +
    distance_dir_global.1 * lup.1 + distance_dir_global.2 * lup.2
output distance_dir := (scale_a / threshold , scale_b / threshold)

output min_distance := d_line.aggregate(over_instances: fresh,
    using: min).defaults(to: 0.0)
output violated := min_distance > threshold
output long_violation @2Hz := violated.aggregate(over_exactly:25s,
    using:forall).defaults(to: false)

trigger violated "warning:Return to flight path!"
trigger long_violation "violation:Countermeasures started!"
```

C The Geofence Specification

```
import math
input x : Float32
input y : Float32
input altitude : Float32
input heading : Float32
input velocity : Float32

// Goefence points given at start of monitor
input p0_x: Float32
input p0_y: Float32
input p1_x: Float32
input p1_y: Float32

// Knots to m/s
output velocity_ms := velocity * 0.514444
constant c_epsilon : Float32 := 0.0000001

// Computes the vehicle line
output diff_x := x - x.offset(by: -1).defaults(to: x)
output diff_y := y - y.offset(by: -1).defaults(to: y)
output isFnc := diff_x != 0.0
output m := if isFnc then (diff_y) / (diff_x) else 0.0
output b := if isFnc then y-(m*x) else 0.0
output dstToPnt := sqrt(diff_x**2.0 + diff_y**2.0)

// true -> going into negative x direction; false -> positive x
output o_x := if abs(diff_x) < c_epsilon then false else diff_x < 0.0
output o_y := if abs(diff_y) < c_epsilon then false else diff_y < 0.0


// Convert geofence points to cartesian coordinates
output m_line(x0, y0, x1, y1)
    spawn with (p0_x, p0_y, p1_x, p1_y)
    eval @p0_x with (y1 - y0) / (x1 - x0)

output b_line(x0, y0, x1, y1)
    spawn with (p0_x, p0_y, p1_x, p1_y)
    eval @p0_x with y1 - (m_line(x0, y0, x1, y1) * x1)

output intersecting(x0, y0, x1, y1)
    spawn with (p0_x, p0_y, p1_x, p1_y)
    eval with m != m_line(x0, y0, x1, y1).hold(or: 0.0) || b =
        b_line(x0, y0, x1, y1).hold(or: 0.0)

output intersection_x(x0, y0, x1, y1)
    spawn with (p0_x, p0_y, p1_x, p1_y)
    eval when isFnc && intersecting(x0, y0, x1, y1) with (b -
        b_line(x0, y0, x1, y1).hold(or: 0.0)) / ( m_line(x0, y0, x1,
        y1).hold(or: 0.0) - m)

output intersection_y(x0, y0, x1, y1)
    spawn with (p0_x, p0_y, p1_x, p1_y)
    eval when isFnc && intersecting(x0, y0, x1, y1) with m_line(x0,
        y0, x1, y1).hold(or: 0.0) * intersection_x(x0, y0, x1, y1) +
        b_line(x0, y0, x1, y1).hold(or: 0.0)

output inbounds(x0, y0, x1, y1)
    spawn with (p0_x, p0_y, p1_x, p1_y)
    eval when isFnc && intersecting(x0, y0, x1, y1) with if x0 < x1
        then (intersection_x(x0, y0, x1, y1) >= x0 and
        intersection_x(x0, y0, x1, y1) <= x1)
    else (intersection_x(x0, y0, x1, y1) <= x0 and
        intersection_x(x0, y0, x1, y1) >= x1)

output in_direction(x0, y0, x1, y1)
    spawn with (p0_x, p0_y, p1_x, p1_y)
    eval when isFnc && intersecting(x0, y0, x1, y1) with if o_x then
        intersection_x(x0, y0, x1, y1) < x else intersection_x(x0,
        y0, x1, y1) > x

output violations(x0, y0, x1, y1)
    spawn with (p0_x, p0_y, p1_x, p1_y)
    eval when isFnc && intersecting(x0, y0, x1, y1) &&
        in_direction(x0, y0, x1, y1) && inbounds(x0, y0, x1, y1)
        with (intersection_x(x0, y0, x1, y1), intersection_y(x0, y0,
        x1, y1))

output inside eval when isFnc with
    violations.aggregate(over_instances: fresh, using: count) % 2 ==
    1

output distance_to_violations(x0, y0, x1, y1)
    spawn with (p0_x, p0_y, p1_x, p1_y)
    eval when isFnc && intersecting(x0, y0, x1, y1) &&
        in_direction(x0, y0, x1, y1) && inbounds(x0, y0, x1, y1)
        with sqrt((intersection_x(x0, y0, x1, y1) - x)**2.0 +
        (intersection_y(x0, y0, x1, y1) - y)**2.0)

output nvl eval when isFnc with
    distance_to_violations.aggregate(over_instances: fresh, using:
    argmin).defaults(to: (0.0,0.0,0.0,0.0))
output next_violation_x eval when isFnc with intersection_x(nvl.0,
    nvl.1, nvl.2, nvl.3).hold(or: 0.0)
output next_violation_y eval when isFnc with intersection_y(nvl.0,
    nvl.1, nvl.2, nvl.3).hold(or: 0.0)
```

```
output next_violation eval when isFnc with (next_violation_x,
    next_violation_y)

output distance_to_violation eval when isFnc with
    distance_to_violations(nvl.0, nvl.1, nvl.2, nvl.3).hold(or: 0.0)

output time_to(x0, y0, x1, y1)
   spawn with (p0_x, p0_y, p1_x, p1_y)
   eval when isFnc && intersecting(x0, y0, x1, y1) &&
       in_direction(x0, y0, x1, y1) && inbounds(x0, y0, x1, y1)
       with distance_to_violations(x0, y0, x1, y1) / velocity_ms

output time_to_violation eval when isFnc with time_to(nvl.0, nvl.1,
    nvl.2, nvl.3).hold(or: 0.0)

trigger eval when isFnc && !inside with "violation:Outside of
    geofence. Return to flight area immediately"
```

References

1. The Crazyflie flying development platform. https://www.bitcraze.io/products/crazyflie-2-1-plus/, accessed: 2025-09-09
2. Baumeister, J., Correnson, A., Finkbeiner, B., Scheerer, F.: An intermediate program representation for optimizing stream-based languages. In: Piskac, R., Rakamaric, Z. (eds.) Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part III. Lecture Notes in Computer Science, vol. 15933, pp. 393–407. Springer (2025). https://doi.org/10.1007/978-3-031-98682-6_20, https://doi.org/10.1007/978-3-031-98682-6_20
3. Baumeister, J., Finkbeiner, B., Kohn, F., Löhr, F., Manfredi, G., Schirmer, S., Torens, C.: Monitoring unmanned aircraft: Specification, integration, and lessons-learned. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II. Lecture Notes in Computer Science, vol. 14682, pp. 207–218. Springer (2024). https://doi.org/10.1007/978-3-031-65630-9_10, https://doi.org/10.1007/978-3-031-65630-9_10
4. Baumeister, J., Finkbeiner, B., Kohn, F., Scheerer, F.: A tutorial on stream-based monitoring. In: Platzer, A., Rozier, K.Y., Pradella, M., Rossi, M. (eds.) Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part II. Lecture Notes in Computer Science, vol. 14934, pp. 624–648. Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_33, https://doi.org/10.1007/978-3-031-71177-0_33
5. Baumeister, J., Finkbeiner, B., Kruse, M., Schwenger, M.: Automatic optimizations for stream-based monitoring languages. In: Deshmukh, J., Nickovic, D. (eds.) Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12399, pp. 451–461. Springer (2020). https://doi.org/10.1007/978-3-030-60508-7_25, https://doi.org/10.1007/978-3-030-60508-7_25
6. Baumeister, J., Finkbeiner, B., Schirmer, S., Schwenger, M., Torens, C.: Rtlola cleared for take-off: Monitoring autonomous aircraft. In: Lahiri, S.K., Wang, C.

(eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12225, pp. 28–39. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_3, https://doi.org/10.1007/978-3-030-53291-8_3

7. Biewer, S., Finkbeiner, B., Hermanns, H., Köhl, M.A., Schnitzer, Y., Schwenger, M.: Rtlola on board: Testing real driving emissions on your phone. In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12652, pp. 365–372. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_20, https://doi.org/10.1007/978-3-030-72013-1_20