Assumption Monitoring of Temporal Task Planning Using Stream Runtime Verification*

Felipe Gorostiaga[1], Sebastián Zudaire[2], César Sánchez[1], Gerardo Schneider[3], and Sebastian Uchitel[4]

[1] IMDEA Software Institute, Spain
[2] Instituto Balseiro - Univ. Nacional de Cuyo, Argentina
[3] Univ. of Gothenburg, Sweden
[4] Univ. de Buenos Aires, Argentina and Imperial College London, UK

Abstract. Temporal task planning uses formal techniques such as reactive synthesis to guarantee that a robot will succeed in its mission. This technique requires certain explicit and implicit assumptions and simplifications about the operating environment of the robot, including its sensors and capabilities. A robot executing a plan can produce a *silent mission failure*, where the user may believe that the mission goals were achieved when instead the assumptions were violated at runtime. This entails that mitigation and remediation opportunities are missed.
Monitoring at runtime can detect complex assumption violations and identify silent failures, but such monitoring requires the ability to describe and detect sophisticated temporal properties together with quantitative and complex data. Additional challenges include (1) ensuring the correctness of the monitors and a correct interplay between the planning execution and the monitors, and (2) that monitors run under constrained environments in terms of resources.
In this paper we propose a solution based on stream runtime verification, which offers a high-level declarative language to describe sophisticated monitors together with guarantees on the execution time and memory usage. We show how monitors can be combined with temporal planning not only to monitor assumptions but also to support mitigation and remediation in UAV missions. We demonstrate our approach both in real and simulated flights for some typical mission scenarios.

1 Introduction

Specifying and developing sophisticated behavior of autonomous systems is a notoriously difficult task. Formal methods is a very tempting approach to the rigorous development of robots and other autonomous systems. Temporal task

* This work was funded in part by the Madrid Regional Government under project "S2018/TCS-4339 (BLOQUES-CM)" and by a research grant from Nomadic Labs and the Tezos Foundation.

planning consists of defining a task specification of express complex robot behaviors, for example, using a fragment of the linear temporal logic (LTL), as we do in this paper. The LTL specification is then transformed into a system using controller synthesis, which implements an operational strategie that guarantees to achieve the desired task [7]. The limitations of current synthesis technology require several modeling simplifications be able to encode the robot, its environment and the mission goals. Consequently, the correctness of the approach depends not only on the correctness of the specification and the synthesis tools used, but also on certain explicit and implicit assumptions about the robot's operating environment, sensors and capabilities. A robot executing a plan can *silently fail* to fulfill the task if the assumptions are violated at runtime, leading the user to mistakenly believe that a mission has been accomplished successfully, which can have significant practical consequences. Sometimes the assumptions can fail due to unforeseen circumstances. Also, the current synthesis technology that is typically used for temporal planning requires a discretization and simplification of the environment that may not always be respected at runtime.

Monitoring assumption violations at runtime can detect silent failures and provide mitigation and remediation opportunities. In this paper we show how temporal task planning can be combined with stream runtime verification (SRV) to monitor both explicit and implicit assumptions on which those plans rely. More concretely, we describe a method to detect and predict assumption violations at runtime to further improve the assurance of robotic behavior.

We use a language and system based on the SRV language Lola to describe the monitors. Lola is a good language for this goal because:

- it has clean semantics,
- its expressive yet succinct syntax results in natural specifications,
- the language allows easy extensions (like new datatypes) and I/O of rich events, and
- it allows the creation of predictive functions such as Kalman filters.

Additionally, Lola allows a theoretical analysis of monitor specifications, which lets us calculate beforehand the constant amount of memory required to run the monitors throughout the entire execution independently of the length of the trace.

We integrate SRV and temporal task planning for two typical mission scenarios of an Unmanned Aerial Vehicle (UAV). We show a simplified architecture of the interplay between the components from temporal task planning and runtime verification in Fig. 1. The mission planning using temporal task problem involves the description of both a temporal task specification and a Lola specification. From these specifications we synthesize a controller and a monitor which communicate with the motion planner, with the vehicle, and with each other through sensors and actuators. We have implemented the UAV controller using MAVLink [2] and we demonstrate our approach empirically both in real flights using the Parrot AR.Drone 2.0 with an onboard mounted Raspberry Pi Zero W and in flights simulated using the Software in the Loop ArduPilot Simulator [1], applying offline and online monitoring. In this paper we focus on the Stream

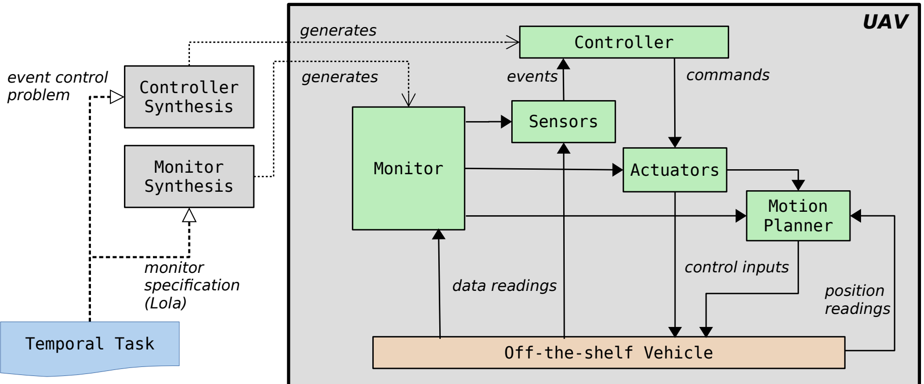


Fig. 1: Architecture for SRV + UAV Temporal Planning.

Runtime Verification part of the approach, and we explain the specifications of the properties to monitor. More details on the mission planning stage and the interaction between the UAV controller and the monitors are available in [11].

The rest of the paper is structured as follows. Section 2 revisits stream runtime verification and the language Lola. Section 3 explains the concept of implicit assumptions and silent mission failures, presenting multiple examples. Section 4 describes an empirical evaluation. Finally, Section 5 concludes.

2 The Stream Runtime Verification Language Lola

We briefly revisit stream runtime verification for synchronous streams. A *synchronous stream* is a sequence of values from a data domain. We refer to the value at the n-th position in a sequence z as $z(n)$. For example, the sequence $altitude = [350, 360, 289, 320, 330]$ contains the successive values of the altitude of an unmanned aerial vehicle. In this sequence $altitude(0) = 350$ and $altitude(2) = 289$. This sequence can be obtained, for example, by sampling a sensor of the UAV every five seconds.

Streams are typed using *data theories*, which are essentially sets of values and interpreted function symbols. The function symbols are used to build stream expressions, and are equipped with an associated implementation that allows computing a result value once the values of the arguments are provided.

Lola is a synchronous SRV language. Specifications (or programs) in Lola declare the relation between output streams (verdicts) and input streams (observations). In this paper we will use an implementation of Lola called HLola[5] (see [5]) written in the purely functional language Haskell. The distinguishing feature of HLola is that data theories are not decided a-priori and hard-wired in

[5] HLola is available at `https://github.com/imdea-software/hlola`.

the tool. Instead, HLola borrows (almost) arbitrary Haskell datatypes as data theories in Lola [6] in a way that is transparent to the monitor engine. We will introduce the language and the implementation concisely. See [3,4] for rigorous formal descriptions of Lola and [5] for the implementation details of the tool HLola.

2.1 Lola Syntax and Semantics

Syntax. A Lola specification uses a set of stream variables which are used to represent inputs and output streams. An output stream is defined by associating it with a defining stream expression, which consists of:

- *offsets* v`[`k`|`d`]` where v is a stream variable of type D, k is an integer number and d a value from D, and
- *function applications* $f\ t_1 \ldots t_n$ in which f is applied to stream expressions $t_1, \ldots, t_n$ of the right types.

Constants are 0-ary function symbols. A stream variable v represents a stream of values of the type of v. If v is an input stream, the values will be provided incrementally, and if v is an output stream the values will be computed by the monitor incrementally. An offset expression v`[`-1`|`d`]` captures the value of sequence v in the previous position. If there is no previous instant, the value d is returned instead. The particular case of an offset with $k = 0$ requires no default value as the index is always guaranteed to be within the range of the sequence, in which case we use v**`[now]`**. A Lola specification consists of a set I of typed input stream variables, a set O of typed output stream variables, and a set of defining equations, $y_i = e_i$, one per output variable $y_i \in O$ where e_i is a stream expression of the same type as y_i. Note that expression e_i can use any stream variables from I and O, including y_i. The defining equations describe the relation between input and output streams. Note that functions f used to build expressions do not need to know about time or offsets. Examples of these function include $\wedge$, $\vee$, $+$, vector multiplication, etc. At runtime, the implementation of theses functions will provide values for the temporal engine to evaluate. This is the essence of stream runtime verification where the temporal reasoning and the algorithms to compute values are kept separate.

Example 1. The specification "*the mean altitude in the last 3 instants*" can be expressed in Lola as follows, where `denom` calculates the number of instants that are taken into account to compute the average.

```
1 input Double altitude
2 output Double denom = min 2 denom[-1|0] + 1
3 output Double mean =
4     (altitude[-2|0] + altitude[-1|0] + altitude[now]) / denom[now]
5 output Bool ok_altitude = mean[now] < 400
```

In this specification $I = \{\texttt{altitude}\}$ and $O = \{\texttt{denom}, \texttt{mean}, \texttt{ok_altitude}\}$. Also, `altitude`, `denom` and `mean` are streams of `Double` values, while `ok_altitude` is a stream of `Bool` values.

Semantics. Semantics of Lola specifications are given in terms of valuations. A valuation ρ consists of the assignment of one sequence ρ_x for each stream variable x, all of the same length. The semantics of Lola is defined in terms of valuations of expressions $[\![e_x]\!]$ as follows:

– For offsets: $$[\![v\texttt{[}i\texttt{|}c\texttt{]}]\!](j) = \begin{cases} \rho_v(j+i) & \text{if } 0 \leq j+i < L \\ c & \text{otherwise} \end{cases}$$

– For functions: $$[\![f\ e_1 \ldots e_k]\!](j) = f\ [\![e_1]\!](j) \ldots [\![e_k]\!](j)$$

Note that f on the left hand side of the semantic definition of a function represents the syntactic representation of the function (the constructor of expressions) while the f on the right hand side represents the function evaluator.

We say that a valuation ρ satisfies a Lola specification φ whenever every output variable y satisfies its defining equation e_y, i.e. when $\rho_y = [\![e_y]\!]$. See [4] for a more rigorous explanation of valuations, where a simple graph traversal of the specifications is used to guarantee correct semantics and to statically compute bounds of the resources needed to perform the monitoring.

HLola: A Powerful Implementation of Lola. HLola is an implementation of Lola developed as an embedded language in Haskell with an extra layer of syntactic facilities. This embedding, called lift deep embedding, extends the notion of eDSL and is described in [3]. This implementation strategy allows us to incorporate constructs from the host language Haskell into Lola, such as let-bindings, where clauses, partial application, list comprehension, etc. We can also use higher order to seamlessly define parametric streams, and we give access to the modules system of Haskell to allow the separation of independent pieces of code. An HLola specification can import an arbitrary Haskell module as well as a Lola *theory* (the implementation of types and functions for a specific domain) or *library* (auxiliary stream definitions of general purpose) with the directive **`use`** at the beginning of the specification.

The powerful type system of Haskell ensures the type-correctness of a specification as well as its syntactic well-formedness. Extra checks specific to the Lola language such as the validity of the dependency graph are assessed by a routine in the HLola engine prior to the monitor execution.

The main function that implements the monitoring algorithm is *runSpec*, which takes an HLola specification and an input trace for every **`input`** stream and returns the successive events of the **`output`** streams. As a by-product of developing HLola as an eDSL in Haskell, the datatypes that constitute an HLola specification and the function *runSpec* are defined in Haskell as well, and as such can be lifted to become a theory ready to be used as a data theory

of HLola itself. This has enabled the use of HLola specifications as a theory from within the language [6].

We describe now this HLola feature, that simplifies specifications in many occasions by allowing the evaluation of data as the result of the computation of a monitor, in order to use the results in higher monitoring activities. We call this feature nested monitoring or nested specifications. Nested specifications allow spawning and executing monitors dynamically, collecting the result of a stream in each invocation and using it as a value in the caller monitor. Defining an *inner* specification involves giving it a name and adding an extra clause: **return** x **when** y where x is a stream of any type and y is a **Boolean** stream. The type of the stream x determines the type of the value returned when the specification is invoked dynamically. Optionally, the definition of a nested specification can require parameters, which can be different in each invocation. Once we have defined a nested specification, we can execute it using the function *runSpec* from the theory **Lola**, importing the theory and providing the necessary parameters and lists of values for the input streams as lists of values of the corresponding types, in the order in which they are defined in the nested specification.

When a nested specification with a return clause **return** x **when** y is executed, the computation will return the value of the stream x at the first time y becomes *True*, or the last value of x if y never holds in the execution. As a consequence, if y becomes *True* in the middle of an execution, the monitor does not have to run until the end to compute a value and can anticipate the result. This opens the door to evaluate the nested specification incrementally and return the outcome as soon as the outcome is definitive. If the specification contains no input stream, then it is executed until the **return** stream becomes *True*.

Example 2. Consider the following specification, which calculates whether input numeric streams **p** and **q** have crossed in the last 50 instants. We define a topmost specification as follows:

```
1  use theory Lola
2  use innerspec crossspec
3  input Double p
4  input Double q
5  output [Double] last50 <Stream Double str> = let
6    prev = take 49 (last50 str[-1|[]])
7    in str[now] ++ prev
8  output Bool haveCrossed = let
9    ps = last50 p
10   qs = last50 q
11   in runSpec (crossspec ps[now] qs[now])
```

The output stream **haveCrossed** invokes the nested specification **crossspec** with the lists of the last 50 events of **p** and **q** as inputs. We use the auxiliary Haskell

function **take** to keep the first 49 elements of the previous list in the variable `prev`, to which we append the corresponding new value.

The inner specification uses the Haskell function **signum** –which returns −1, 0, or 1, indicating the sign of the argument– to check that the sign of the difference is maintained throughout the entire subtrace:

```
1 innerspec Bool crossspec
2 input Double p
3 input Double q
4 output Bool cross =
5   signum (p[now] - q[now]) /== signum (p[-1|p[now]] - q[-1|q[now]])
6 return cross when cross
```

Note that the inner monitor returns as soon as the signals are detected to cross.

3 Implicit Assumptions and Silent Mission Failures

Temporal task planning using reactive synthesis requires a discrete description of the goal (and the world in which the system will operate). Creating discrete domains for complex continuous enivironments requires introducing assumptions regarding the relation between the discrete abstraction and the intention in the real world that typically includes continuous variables. Thus, the objective in the *real world* can be informally described as "A_{RW} implies G_{RW}" (where A_{RW} represents the assumptions in the real world and G_{RW} represents the mission goal in the real world). Reactive synthesis solves this problem by using *discrete* goals and assumptions $A_D \rightarrow G_D$. The correctness of the solution is based on two additional assumptions: "A_{RW} implies A_D" and "G_D implies G_{RW}", in other words that the discrete assumptions hold in the real world and that accomplishing the discrete goals imply accomplising the mission in the real world.

We refer to A_{RW} and the last two implications as *implicit assumptions*. A *silent mission failure* occurs when a system fails to achieve its goals G_{RW} but A_{RW} does not hold either. As a consequence the implication "A_{RW} implies G_{RW}" is satisfied, making it impossible for the system user to distinguish if G_{RW} holds or if A_{RW} did not hold.

3.1 Handling Assumption Violations

We show two scenarios in which we combine temporal task planning and Stream Runtime Verification

(A) Flagging assumption violations after a mission concludes in order to detect silent failures.

(B) Triggering recovery measures online, such as replanning or mission abortion, immediately upon an assumption violation using quantitative streams in specifications to facilitate plan adjustment [6].

We now discuss these separately by describing two missions.

SRV for Offline Monitoring. The first mission uses SRV for assumption monitoring and logging. This is a search and rescue mission with a no-fly zone, with the instruction of landing when the target is found. The monitor processes the incoming events and monitors and logs assumption violations alongside the associated relevant information, which can then be used offline for mission post-hoc analysis.

In this mission we monitor two assumptions. The first assumption checks that the UAV never enters the no-fly zone, and, if it does, measures the degree of violation as the distance inside the no-fly zone. The second assumption ensures that the area sensed when entering a zone covers the zone completely.

The first assumption can be easily monitored by calculating for each input position reading, the depth inside the no-fly zone, returning **Nothing** if the UAV is not within a forbidden zone.

There are several ways one could monitor the second assumption. In this specification we monitor a stronger property that requires the attitude and position of the vehicle to be within certain bounds. Since these data measurements may carry noise we include in the monitor specification a filtering phase that implements a first order low-pass filter. Note that most cameras may require several seconds to capture a high resolution image, specially on low-end hardware. For this reason, it is necessary to monitor that the filtered attitude and position is within the required bounds for every instant between the `open_capture` command and its corresponding response event, which is the time frame in which a picture is being taken. Below we show an HLola specification that encodes both assumptions.

[6] The complete discrete event control problems and assumption used to synthesise plans and monitors, and the data obtained from the real and simulated flights discussed in this section is available at `http://mtsa.dc.uba.ar`.

```
1 use theory Geometry2D
2 use haskell Data.Maybe
3 use haskell Data.List
4 data Attitude = Attitude {yaw ::Double, roll ::Double, pitch ::Double}
5 data Target   = Target {x :: Double, y :: Double, num_wp :: Double}
6 data Position = Position {x :: Double, y :: Double, alt :: Double}
7 type Vector = (Double, Double)
8 type Position = (Double, Double)
9 type Polygon = [(Double, Double)]
10 input Attitude  attitude
11 input Vector    velocity
12 input Position  position
13 input Double    altitude
14 input Target    target
15 input [Polygon] noflypolys
16 input [String]  events_within
17 output Bool all_ok_capturing = capturing[now] ‘implies‘
18   (height_ok[now] && near[now] && roll_ok[now] && pitch_ok[now])
19 output Bool flying_in_safe_zones = flying_in_poly[now] === Nothing
20 output (Maybe Polygon) flying_in_poly = let
21   position_in_poly = pointInPoly filtered_pos[now]
22   in find position_in_poly noflypolys[now]
23 output (Maybe Double) depth_into_poly = let
24   mSides = funmap polygonSides flying_in_poly[now]
25   distance_from_pos = shortestDist filtered_pos[now]
26   in funmap distance_from_pos mSides
27   where
28   shortestDist x = minimum (map distancePointSegment x)
29   funmap f Nothing = Nothing
30   funmap f (Just x) = Just (f x)
31 output Bool capturing = let
32   has_capture = "capture" ‘belongsto‘ events_within[now]
33   in has_capture || open_capture [-1|False]
34 output Bool open_capture =
35   lastIsCapture open_capture[-1|False] events_within[now]
36   where
37   lastIsCapture dflt [] = dflt
38   lastIsCapture _ xs = last xs == "capture"
```

```
40 output Double f_pos_component <(Position->Double) field> = let
41   tau     = 0.6
42   _this   = f_pos_component field
43   in (field position[now] + tau*(2*_this[-1|0] - _this[-2|0]/2))
44      / (1 + 1.5*tau)
45 output Double filtered_x   = (f_pos_component x)[now]
46 output Double filtered_y   = (f_pos_component y)[now]
47 output Double filtered_alt = (f_pos_component alt)[now]
48 output Position filtered_pos = (filtered_x[now], filtered_y[now])
49 output Bool near = distance filtered_pos[now] target[now] < 1
50 output Bool height_ok = filtered_alt[now] > 0
51 output Bool roll_ok  = abs (roll attitude[now]) < 0.0523
52 output Bool pitch_ok = abs (pitch attitude[now]) < 0.0523
```

The specification imports geometric facilities from **theory** `Geometry2D`, and Haskell libraries `Data.Maybe` and `Data.List`, and defines custom datatypes to interpret the data from the UAV, in lines 1–9. The input streams of this specification, declared in lines 10–16 encompass the state of the UAV, the current target position, the no-fly zones and the onboard camera events to detect when a picture is being captured. The output stream **`all_ok_capturing`** declared in line 17 assesses that, whenever the vehicle is taking a picture, the height, roll and pitch are acceptable and the vehicle is near the target location. The output stream **`flying_in_safe_zones`** from line 19 reports if the UAV is flying outside the forbidden regions. The stream **`flying_in_poly`** in line 20 partially applies the function pointInpoly from the theory `Geometry2D` to the current position to find and return the forbidden region in which the vehicle is flying, if any, using the Haskell function **find**. The output stream **`depth_into_poly`** defined in line 23 takes the minimum of the distances between the vehicle position and every side of the forbidden region inside of which the vehicle is flying, if any.

The intermediate stream **`capturing`** from line 31 expresses whether the UAV is taking a picture, using the auxiliary stream **`open_capture`** declared in line 34. The streams **`filtered_alt`** and **`filtered_pos`** from lines 47 and 48 represent the location and altitude of the UAV filtered to reduce noise from the sensors, a procedure that is implemented generically by the parameterized stream **`filtered_pos_component`** in line 40. The intermediate stream **`near`** declared in line 49 takes the distance from the vehicle location to its target and checks if it is lower than 1. Finally, the streams **`height_ok`**, **`roll_ok`**, and **`pitch_ok`** from lines 49–51 calculate that the corresponding attitude of the vehicle is within certain predefined boundaries.

A logger will store the values of all these streams whenever **`all_ok_capturing`** is *False* to detect a silent failure and allow a post-hoc analysis.

SRV for Online Monitoring and Adaptation. In the second mission we demonstrate how SRV can be used, beyond logging, to support remediation actions during the flight by interacting with components of the UAV controller. We ran an ordered patrol mission (as described in [8]) of three locations with no-fly zones.

We built a monitor that uses quantitative data to *predict* the violation of assumptions, enabling the possibility of reacting to anticipate and avoid the faulty behavior *before* it occurs. To achieve this, the HLola monitor specification encodes a simplified non-linear 2D model of a fixed-wing UAV and the full extent of the waypoint guidance control algorithm of ArduPilot. Part of the input to this monitor includes the current state of the system (position, attitude, wind) and the list of waypoints for the current trajectory. The monitor uses this input together with the non-linear model of the UAV and the guidance control algorithms to produce, by means of simulation into the future, a prediction of the UAV's flight path.

We add an *Extended Kalman Filter* (EKF) to estimate in-flight a parameter *tau* from the simplified UAV models, helping to diminish the error in the predicted flight path. Other parameter identification techniques could be used instead (e.g., least squares), but we aimed to show HLola's ability to implement state-of-the-art estimation algorithms.

The monitor produces an event to indicate that the monitor anticipates an abnormal situation in the future:

- If the UAV is predicted to hit the target zone but far from its center point, then a nearMiss event is produced.
- If the target zone is to be completely missed, an event goFailed is produced. Our monitor is specified to hold its failure prediction for several time instants before producing a goFailed, to allow the controller to attempt to correct the airspeed.
- Finally, the monitor produces an event abnormalDrift when the predicted distance of the UAV to its intended location is beyond a threshold.

To utilize the output of our predictive monitors, the following three requirements are added to the mission plan:

(1) upon a nearMiss the UAV should reattempt to visit the location immediately,
(2) upon a goFailed the UAV should skip the location and move onto the next location to patrol, and
(3) upon an abnormalDrift event, the UAV should abort mission and land.

We monitor two properties. The first property predicts how close to the center of the target zone the UAV will pass, so that the UAV controller can either abort the mission or try to correct the flight path based on the degree of the error predicted.

The second property observes how the UAV is following the planned path by predicting the estimated time of arrival and total flight distance. A significant difference is likely to be due to wind conditions, which is not considered by

the plan. The monitor uses this estimated arrival time and distance to compute the speed at which the vehicle is more likely to succeed in closely following the trajectory, and the controller uses this information to change the speed of the UAV accordingly.

Below we show the HLola specification that monitors both assumptions.

```
1  use theory Geometry2D
2  use theory SimulatedGuidance
3  use theory GuidanceTheory
4  use haskell Data.Either
5  use theory Lola
6  use innerspec simuguidance
7  type Matrix = [[Double]]
8  input NonLinearData navigation_data
9  input Vector        velocity
10 input Point         position
11 input Point         target
12 input Vector        target_dir
13 output Double intersDistance = let
14   r0 = (position[now], velocity[now])
15   r1 = (target[now], target_dir[now])
16   in intersectionDistance r0 r1
17 output Double pi_controller =
18   saturate (21 + estimated_error[now] * 0.5 + err_int[now])
19   where
20   saturate x = if x<15 then 15 else if x>30 then 30 else x
21 output Double estimated_error =
22   21 - distance estim_data[now] / time estim_data[now]
23 output (Matrix, Double, Double) kfuStream = let
24   inroll      = initial_roll navigation_data[now]
25   prev_iroll  = initial_roll navigation_data[-1| navigation_data[now]]
26   dt          = 0.05
27   dfltM       = [[0.01,0],[0,0.01]]
28   kfp         = kalman_filter_predict navigation_data[now] 1 tau[-1|0.5]
29   in
30   kalman_filter_update
31       matrix_kfiltered[-1| dfltM] prev_iroll tau[-1|0.5]
32       inroll kfp dt
33 output Matrix matrix_kfiltered =
34   fst3 kfuStream[now]
35 output Double tau = let
36   thd3 kfuStream [now]
```

```
37 output ([Point2], Double, Bool, Double, Double) simulated_guidance =
38   let wp_radius_wp = get_wp_radius_wp navigation_data[now]
39   in
40   runSpec (simuguidance navigation_data[now] tau[now] 200 wp_radius_wp)
41   where
42   get_wp_radius_wp (NLD _ wp_list _ _ n_wp radius_list _ _ _ _ _) =
43     filter farEnoughWP
44       (drop n_wp (zip3 wp_list radius_list (Nothing:map Just wp_list)))
45   farEnoughWP (_, _, Nothing) = True
46   farEnoughWP (nwp, _, Just prev_wp) = norm (nwp ‘minus‘ prev_wp)>0.01
47 output [Point2] simulated_guidance_wps = let
48   fst5 simulated_guidance[now]
49 output Estimation estim_data =
50   simulate_guidance tau[now] navigation_data[now]
51 output Double err_int = if isSaturated pi_controller[-1|21] then
52   err_int [-1|0] else err_int [-1|0] + estimated_error [now] * 0.03
53   where
54   isSaturated x = x==15 || x==30
```

The input streams are data about the state of the vehicle at every instant (**navigation_data**, **velocity** and **position**) and its current target destination (**target** and **target_dir**).

The output stream **intersDistance** of line 13 estimates how far from the target location the UAV will pass, returning -1 if the UAV is actually moving away from the target location. The output stream **pi_controller** shown in line 17 is used to control the UAV's airspeed set-point, taking a reference velocity of 21 meters per second and an estimation of the average UAV speed along with the predicted flight path, generated from the predicted arrival time and flight distance, which is calculated by the stream **estim_data**, declared in line 49. The main idea is that in order to correctly follow the planned path one must satisfy a constant turn radius assumption, which (for the UAV) translates into constant ground-speed. When the UAV has tailwind, for a constant airspeed, it will fly at a faster ground-speed than when it experiences headwind. Controlling the ground-speed around a fixed value (by actuating on the airspeed set-point) helps following the desired path more accurately.

The intermediate stream **kfuStream** shown in line 23 implements the EKF at every instant, and the intermediate stream **simulated_guidance** from line 37 simulates the UAV trajectory based on the current **navigation_data**, the **tau** (provided by the EKF and extracted in line 35), a maximum flight time of 200 and the target path, given by the list of its waypoints.

The function simulate_guidance from the theory **GuidanceTheory** uses a nested specification **simuguidance** to simulate the trajectory of the UAV over the current list of waypoints, which is captured in the input stream **route**.

```
1  innerspec ([Point2], Double, Bool, Double, Double) simuguidance
2                       <NonLinearData nld> <Double tau> <Double maxtime>
3  use theory GuidanceTheory
4  use theory Geometry2D
5  use theory Lola
6  use haskell Data.List.Extra
7  use innerspec simuinner
8  type Route = (Point2, Double, Maybe Point2)
9  input Route route
10 output ReachStateType reaching_state = let
11   roll = reach_roll [-1|initial_roll nld]
12   yaw = reach_yaw [-1|initial_yaw nld]
13   distance = reach_distance [-1|0]
14   time = reach_time [-1|0]
15   pos = reach_pos [-1|initial_pos nld]
16   in
17   runSpec
18     (simuinner nld tau maxtime pos yaw roll time distance route[now])
19 output Bool reached_maxtime = reached_maxtime_field reaching_state[now]
20 output Double reach_time = reach_time_field reaching_state[now]
21 output Double reach_distance = reach_distance_field reaching_state[now]
22 output Double reach_yaw = reach_yaw_field reaching_state[now]
23 output Double reach_roll = reach_roll_field reaching_state[now]
24 output Point2 reach_pos = reach_pos_field reaching_state[now]
25 output [Point2] wps =
26   snoc wps[-1| []] reach_pos[now]
27   where
28   snoc ls x = ls ++ [x]
29 output Bool has_reached = reached_maxtime [-1|False]
30 output ([Point2], Double, Bool, Double, Double) ret =
31   (wps[now], reach_roll[now], reached_maxtime[now],
32     reach_time[now], reach_distance[now])
33 return ret when has_reached
```

The successive values of **route** are the linear trajectories that the UAV is expected to go through. The monitor **simuguidance** will run until the route is covered or until the **maxtime** is exceeded. For every part of the route, **simuguidance** will spin up a second level nested specification **simuinner** to simulate successive small steps of the flight in a path between two waypoints.

```
1 innerspec ReachStateType simuinner <NonLinearData nld> <Double tau>
2   <Double maxtime> <Point2 dpos> <Double dyaw> <Double droll>
3   <Double dtime> <Double ddistance> <(Point2, Double, Maybe Point2) wps>
4 use theory GuidanceTheory
5 use theory Geometry2D
6 data ReachStateType = RST { reached_maxtime_field :: Bool,
7                             reach_time_field :: Double,
8                             reach_distance_field :: Double,
9                             reach_yaw_field :: Double,
10                            reach_roll_field :: Double,
11                            reach_pos_field :: Point2}
12 const dt = 0.05
13 const g = 9.8
14 const lim_roll_cd = 65 * pi / 180
15 const limit = sin lim_roll_cd
16 const NLD navl1_d _ navl1_p _ _ _ _ gamma vel_a vel_w _ = nld
17 const (dposx, dposy) = dpos
18 const (next_wp, wp_radius, mprev_wp) = wps
19 output Double time = step_time [-1|dtime]
20 output Double step_time = time[now] + dt
21 output Double yaw = step_yaw [-1|dyaw]
22 output Double step_yaw = let
23   yaw_dot = (g * (sin (roll[now]))) / vel_a
24   in yaw[now] + dt * yaw_dot
25 output Double posx = step_posx [-1|dposx]
26 output Double step_posx =
27   posx[now] + dt * (vel_a * sin yaw[now] + (vel_w * sin gamma))
28 output Double posy = step_posy [-1| dposy]
29 output Double step_posy =
30   posy[now] + dt * (vel_a * cos yaw[now] + (vel_w * cos gamma))
31 output Point2 pos = P posx[now] posy[now]
32 output Double roll = step_roll [-1|droll]
33 output Double step_roll = let
34   vel_g = (fst.get_ground_speed vel_w vel_a gamma) yaw[now]
35   hdg = (snd.get_ground_speed vel_w vel_a gamma) yaw[now]
36   gcp = guidance_control_parameters navl1_d navl1_p next_wp
37                                     prev_wp[now] vel_g hdg pos[now]
38   l1 = fst3 gcp
39   k_l1 = snd3 gcp
40   eta = thd3 gcp
41   demanded_roll = (asin.max (-limit).min limit)
42                      ((k_l1 * vel_g * vel_g * sin eta) / (l1 * g))
43   in
44   (dt * demanded_roll + tau * roll[now]) / (tau + dt)
```

```
43 output Double distance = step_distance [-1|ddistance]
44 output Double step_distance = let
45   step_pos = (step_posx[now], step_posy[now])
46   in distance[now] + norm (step_pos ‘minus‘ pos[now])
47 output Point2 prev_wp = maybe pos[now] Leaf mprev_wp
48 output Bool reached_maxtime = time[now] >= maxtime && maxtime > 0
49 output Bool stop_simulation = let
50   vnwp = verify_next_waypoint next_wp wp_radius prev_wp[now] pos[now]
51   in reached_maxtime[now] || vnwp /== INTRAVEL
52 output ReachStateType ret = RST reached_maxtime[now] time[now]
53                               distance[now] yaw[now] roll[now] pos[now]
54 return ret when stop_simulation
```

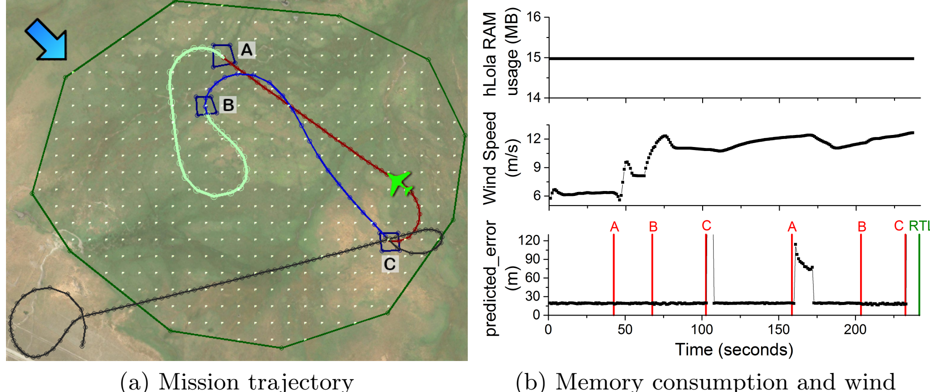
Every execution of **simuinner** updates the UAV roll, yaw, and position, the time consumed, and the distance reached in the simulation step. The nested monitor **simunner** has no input stream and as a consequence it will execute until **stop_simulation** becomes *True*. The monitor calculates the trajectory of the UAV updating its position, distance traveled, roll, and yaw according to simulation steps of 0.05 time units. The simulation stops when the **maxtime** is reached, or when the UAV is no longer `INTRAVEL` and returns the latest state of the UAV, along with the total time consumed and the information of whether the maximum time was reached.

4 Empirical Evaluation

Our UAV control system (including the HLola monitors) was built for MAVLink [2] complying aerial vehicles, and tested extensively on the ArduPilot Software-In-The-Loop (SITL) simulator as in [9]. We have used this simulator (see [10]) to seamlessly transition from simulation to actually flying a fixed-wing vehicle based on a Pixhawk. The controller was run on a single Raspberry Pi.

To evaluate if the UAV entered no-fly zones during its mission using offline monitoring, we have carried out real flights using the Parrot Ar.Drone 2.0 and an onboard mounted Raspberry Pi Zero W, which we have used to log the flight data. The HLola monitor used a constant amount of memory of around 13.4 megabytes during the whole post-flight analysis.

The mission that uses online monitoring to predict and correct the behavior of the UAV was simulated on the ArduPlane SITL simulator, with a Raspberry Pi 3B+ as the simulated hardware, just like in [10]. We show the actions taken by the monitor during the patrol mission in Fig. 2 (a). We have incremented the simulated wind speed over time to mimic an adverse situation and trigger the monitor, as we show in Fig. 2 (b). By doing so, the second time the UAV travels from A to B, the predicted arrival distance error rises over 100 meters, triggering the calculation of a new trajectory until one that guarantees the correct arrival



(a) Mission trajectory (b) Memory consumption and wind

Fig. 2: Mission trajectory and flight data

of the UAV to the location is found. After the second time C is visited, the wind is too strong and the UAV is commanded to return to the launch point. Fig. 2 (b) also illustrates the memory consumption of the monitor, which remains practically constant as expected (see [3] for details about the memory consumption of HLola).

Our experiments show how we can monitor assumptions for UAV missions using HLola with negligible interference with the main navigation system, in particular with constant memory consumption and in constant time per event, providing support for the hypotheses that motivated the use of SRV and HLola.

5 Conclusion

We have applied the infrastructure from Runtime Verification to monitor assumptions in the context of temporal task planning, both offline for post-hoc analysis and online to enable remediation actions.

We have been able to describe the assumptions with HLola, a very expressive language for Stream Runtime Verification, using multi-level specifications to predict the behavior of the vehicle using complex estimators. We have automatically synthesized the monitors that collect and process sophisticated data, generating rich verdicts while providing strong correctness and resource consumption guarantees.

We are currently working on a deeper interplay of monitoring and planning in an architecture that allows modular and flexible definitions of widely diverse UAV missions, reusing monitors across different tasks, and leveraging on the formal semantics and guarantees from the field of Stream Runtime Verification to further improve the reliability of robots running temporal plans.

References

1. SITL/ardupilot simulator (software in the loop). http://ardupilot.org/, accessed: 2021-10-20
2. Atoev, S., Kwon, K., Lee, S., Moon, K.: Data analysis of the mavlink communication protocol. In: ICISCT'17. pp. 1–3 (Nov 2017)
3. Ceresa, M., Gorostiaga, F., Sánchez, C.: Declarative stream runtime verification (hLola). In: Proc. of the 18th Asian Symposium on Programming Languages and Systems (APLAS'20). LNCS, vol. 12470, pp. 25–43. Springer (2020). https://doi.org/10.1007/978-3-030-64437-6_2
4. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: Runtime monitoring of synchronous systems. In: Proc. of the 12th Int'l Symp. of Temporal Representation and Reasoning (TIME'05). pp. 166–174. IEEE CS Press (2005)
5. Gorostiaga, F., Sánchez, C.: HLola: a very functional tool for extensible stream runtime verification. In: Proc. of the 27th Int'l Conf on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'21). Part II. pp. 349–356. LNCS, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_18
6. Gorostiaga, F., Sánchez, C.: Nested monitors: Monitors as expressions to build monitors. In: Feng, L., Fisman, D. (eds.) Runtime Verification. pp. 164–183. Springer International Publishing, Cham (2021)
7. Kress-Gazit, H., Fainekos, G., Pappas, G.: Translating structured english to robot controllers. Advanced Robotics **22**, 1343–1359 (10 2008)
8. Menghi, C., Tsigkanos, C., Pelliccione, P., Ghezzi, C., Berger, T.: Specification patterns for robotic missions. IEEE Transactions on Software Engineering pp. 1–1 (2019)
9. d. S. Barros, J., Oliveira, T., Nigam, V., Brito, A.V.: A framework for the analysis of uav strategies using co-simulation. In: SBESC'16. pp. 9–15 (Nov 2016)
10. Zudaire, S.A., Garrett, M., Uchitel, S.: Iterator-based temporal logic task planning. In: 2020 IEEE International Conference on Robotics and Automation (ICRA). pp. 11472–11478 (2020). https://doi.org/10.1109/ICRA40945.2020.9197274
11. Zudaire, S., Gorostiaga, F., Sánchez, C., Schneider, G., Uchitel, S.: Assumption monitoring using runtime verification for UAV temporal task plan executions. In: Proc. of IEEE International Conference on Robotics and Automation (ICRA'21). IEEE (2021)