Bridging Reactive and Control Architectural Layers for Cooperative Missions Using VTOL Platforms

Piotr Rudol and Patrick Doherty
Department of Computer and Information Science
Linköping University Linköping, Sweden
email: piotr.rudol@liu.se

***Abstract*—In this paper we address the issue of connecting abstract task definitions at a mission level with control functionalities for the purpose of performing autonomous robotic missions using multiple heterogenous platforms. The heterogeneity is handled by the use of a common vocabulary which consists of parametrized tasks such as *fly-to*, *take-off*, *scan-area*, or *land*. Each of the platforms participating in a mission supports a subset of the tasks by providing their platform-specific implementations. This paper presents a detailed description of an approach for implementing such platform-specific tasks. It is achieved using a flight-command based interface with setpoint generation abstraction layer for vertical take-off and landing platforms. We show that by using this highly expressive and easily parametrizable way of specifying and executing flight behaviors it is straightforward to implement a wide range of tasks. We describe the method in the context of a previously described robotics architecture which includes mission delegation and execution system based on a task specification language. We present results of an experimental flight using the proposed method.**

I. Introduction

Recent research and development concerning Unmanned Aerial Vehicles (UAVs) has focused strongly on the vehicle airframe and its avionics and sensors. Low-level control systems and navigation functionalities combined with task and motion planners provide the necessary support for performing missions of low to moderate complexity. However, mission specification is often manual or semi-automatic and results in a sequence of actions a robot should perform to fulfill the mission objectives. This is often time consuming and also prone to error due to the low level of abstraction used and the lack of automation in generating plans.

The problem becomes more complex when considering missions involving collaborating systems. Such missions typically include multiple platforms together with operators where interaction between them is necessary to achieve mission goals. This increases the complexity of the functionalities required by individual platforms as well as of the architectural support required for collaboration and task specification.

One way of dealing with the interoperability of heterogenous systems in cooperative missions is to split the definitions of the functional building blocks, the *tasks*, into two parts: Generic high-level declarative task specifications understood by all participating systems, and a set of platform-specific implementations, each of which interfaces to the low-level flight functionalities of a particular platform.

Typical high-level tasks for UAVs include taking off, flying to a sequence of positions, following a target, patrolling a region, and landing. Traditionally, all these actions are seen as atomic at the mission level and are handled using dedicated control functions implementing continuous control laws. The switching between these tasks is performed through transitions in a hybrid automaton which combines continuous control with discrete mode switching.

This approach, however, suffers from a number of shortcomings. First, the transitions between flight functionalities are potentially non-trivial, requiring adding helper states in the automaton to preserve the continuity of control signals. In case of flying robots this usually results in *fly-brake-stop-fly* behavior. Second, the repertoire of existing control functions for performing specific actions or maneuvers is static. Adding new functionalities requires either restructuring and re-parameterizing existing ones or even developing completely new ones which is usually time consuming.

This paper shows how platform-specific execution functionalities for common tasks can be implemented using a flight-command based interface with a setpoint generation abstraction layer for vertical take-off and landing platforms. The use of this particular approach gives the ability to easily implement a wide range of flight behaviors. This includes functionalities where a high level of *interactivity* between a platform and an operator is required. One example of this aspect is a mission *break-in*, which involves suspending the current flight activity, switching to a new one, and eventually resuming the execution of the original mission.

The functionalities described here are fully implemented within the Hybrid Deliberative/Reactive HDRC3 architecture for unmanned aircraft systems [1]. This architecture has been instantiated and used with multiple vertical take-off and landing aerial vehicles such as the Yamaha RMAX helicopter and the LinkQuad quadrotor, but is not limited to use in particular aircraft systems. It combines various generic functionalities essential to the integration of low autonomy and high autonomy in a single system and has also

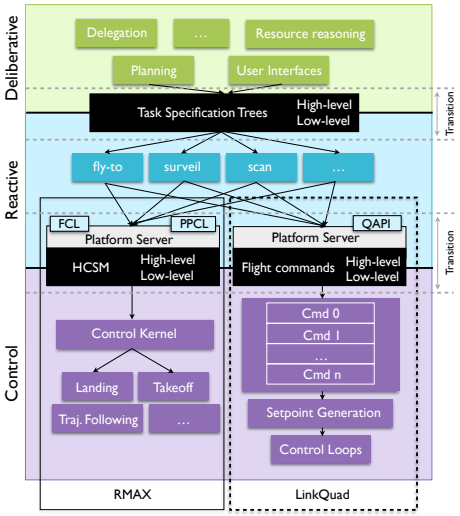


Figure 1: Layered view of the Hybrid Deliberative/Reactive (HDRC3) architecture instantiated for two autonomous platforms: the RMAX helicopter and the LinkQuad quadrotor. The dashed line outlines the focus of this paper.

been extended for use with multi-platform systems. High-level tasks are specified through a Task Specification Tree (TST) formalism that also supports context-dependent and platform-specific implementations through the use of TST Executors, which make use of the flight command interface presented in this paper. TSTs thus provide both a declarative and procedural means of transitioning between the deliberative and reactive layers of a robotic architecture. The focus of this paper within the HDRC3 robotics architecture is highlighted using a dashed rectangle in Fig. 1.

The remainder of the paper is structured as follows. First, the Task Specification Trees are described in Section III with the emphasis put on the implementation aspects of task executors. Sections IV and V follow with the description of the existing low-level control system as well as its Robot Operating System[1] (ROS) based wrapper. Section VI describes implementations of example TST Executors, and finally Section VII describes results of experimental flights using these task executors.

II. Related work

Software architectures for mobile robots generally use three-layered (3T) structures, where each layer has specific temporal properties associated with the length of decision cycles. The first *reactive layer*, with the shortest cycles (orders of milliseconds), consist of a set of reactive skills which map sensor stimuli directly onto actuators with minimal internal state. The second *sequencing layer*, with a larger decision cycle (measured in seconds), is responsible for sequencing of activities. The third layer, with the longest decision cycles measured in seconds or even minutes, is the *deliberative layer*, which is responsible for reasoning about mission goals through the use of search-based techniques [2]. The traditional 3T structure has been extended in a number of ways. For example, a four layer variant has been presented in [3]. A mission management system based on a 3T architecture for a VTOL UAV has been proposed in [4]. The paper presents the overall structure of an architecture and proposes a *supervisory system* that interacts with a *sequence control system*. It, in turn, contains *movement primitives*. The model of the transition between these primitives includes a *slow down* state which brings the system into a *stand by* mode.

An architecture with a more *hybrid* flavor and slightly different naming of the layers, *deliberative, reactive and control* (HDRC3), has been described in [5]. Its general structure is presented in Fig. 1. It uses concepts of delegation and Task Specification Trees described in the following section and in detail in [1].

A considerable amount of research has been devoted to addressing the control issues as well as development of platforms themselves. This has allowed for flying a wide range of maneuvers – from waypoint following [6], through take-offs [7] and landings [8] to, for example, employing machine learning to improve the flight performance over time [9], or for the purpose of avoiding obstacles [10].

III. Delegation and Task Specification Trees

While this paper focuses on the transition between reaction and control, we must first place these contributions in their proper context by briefly discussing certain aspects of the higher deliberative layers in the HDRC3 architecture.

When an unmanned system is viewed as an agent, it is natural to view the assignment of a complex mission to that system as *delegation*, the act of assigning an agent the authority and responsibility to carry out specific activity. The HDRC3 architecture uses the delegation concept for dealing with these issues as well as the problem of task allocation for multiple systems: A *delegator* can ask a *contractor* to take the responsibility for a specific task to be performed under a set of constraints representing mission requirements, and a contractor can in turn ask other agents for assistance by attempting to delegate subtasks to them [1].

Delegation requires a general and expressive task specification language, and HDRC3 uses Task Specification Trees (TSTs) for this purpose. Inner nodes in a TST can specify standardized control structures such as sequences (S), concurrent execution (C), conditionals (IF) and loops (WHILE). Leaf nodes declaratively specify potentially domain-specific tasks to be executed, typically corresponding to high-level actions such as taking off or scanning an area. Such tasks are viewed as elementary and indivisible from the point of view of a delegator, but during the delegation process, contractors can choose to elaborate and *expand* them into trees of

[1]ROS: www.ros.org

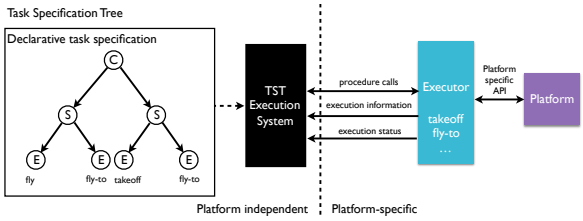


Figure 2: Schematic view of the interaction between Task Specification Trees and platform-specific TST Executors.

subtasks through calls to general task planners or problem-specific functionalities such as scan pattern generators.

Each task in a TST is parameterized and parameters can either be specified directly or constrained through a general constraint language, allowing a certain degree of freedom in setting parameters in order to satisfy the overall requirements of a mission. For example, the flight velocity and altitude for a particular task can be constrained to be within specific intervals, allowing these parameters to be adapted to fuel and time constraints.

A platform can accept the delegation of a TST – a mission or a part of a mission – if it can execute the root node of that TST. It may then also execute the children of the node or may delegate those to other participants.

In order to execute a particular node, a platform requires an *executor* providing a procedural implementation of the declarative specification encapsulated in a TST node. Platforms can share implementations of some executors, such as those corresponding to general control structures. However, most executors must be *platform-specific* in order to call the proper platform-specific functionalities. These executors must satisfy the general definition of the node type in question, such as *fly-to*, and must also explicitly declare any platform-specific constraints on their parameters. For example, each platform may have a distinct constraint on altitude and velocity.

The TST Execution System (see Fig 2 right) is responsible for interfacing with platform-specific executors. This architectural element includes support for executing and synchronizing tasks in a distributed manner, potentially across several robotic platforms, and is used during the delegation process as well as during the execution of the task.

The details of the involved mechanisms, however, are outside the scope of this work. A detailed description of TSTs and the delegation functionalities used in the HDRC3 robotic architecture is presented in [5]. From the perspective of the work presented in this paper, implementing TST Executors is the most important aspect and therefore the remainder of this section will focus on this issue.

A. Implementing TST Executors

A TST Executor is implemented as a class. Therefore the process of implementing platform-specific executors involves providing program code for a set of predefined methods as well as setting values in certain data structures used for specifying execution properties and status feedback information. Interaction between the TST Execution System and executors is performed in two phases. First, an executor participates in a delegation process. Second, the executor performs the actual execution of a task. An overview of a lifecycle of an executor is presented in Fig. 3.

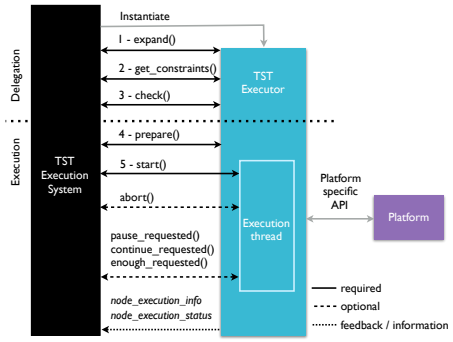


Figure 3: Lifecycle of a Task Specification Tree Executor.

During the delegation process, the TST Execution System invokes the following methods:

- `expand()`
- `get_constraints()`
- `check()`

As already mentioned, leaf nodes specify potentially domain-specific tasks which are viewed as elementary from the point of view of a delegator. Specific implementations, however, may include an expansion of the task into subtasks. This process is invoked by calling the `expand()` method. Its implementation may involve using problem-specific functionalities or calling a task planner to generate additional nodes. These are then returned and appended to the tree, potentially including sequence and/or concurrency nodes.

TSTs provide support for specifying a task's context in form of constraints. Therefore a call to `get_constraints()` returns any constraints (in a general constraint language [1]) which are platform-specific and should be checked for consistency with a constraint solver for both the delegation and execution phases. An example constraint to enforce might be a maximum duration of a task which is limited by the current fuel level. Obtaining this type of information may require interacting with the platform directly to get the most up to date set of constraints. If that is the case, appropriate calls through a platform's API are made.

The last call invoked during the delegation phase is `check()`. Its purpose is to verify whether the execution

can be performed with the provided input parameters as well as checking whether other platform-specific requirements are met. For example, a check whether a camera or a laser range finder sensors are present, properly configured and ready to provide data required for the specific task the executor defines.

Beside implementing the three methods, a TST Executor additionally has to provide execution information by setting fields in a data structure summarized in Fig. 4. The information is used to inform the TST Execution System whether the execution of the task implemented using the node can be aborted (`can_be_aborted`), paused (`can_be_paused`), or stopped and finished (`can_be_enoughed`). In this context, *enoughing* means that the task should stop as *enough* of it has been performed (e.g. scanning of an area). It is distinct from aborting which is considered an abnormal and urgent stopping of a task which should yield an immediate reaction from the platform. The execution information can be set directly based on the intentions of the programmer, properties of the platform, or can be based on the information provided through a platform's API.

Name	Meaning
can_be_aborted	True if execution can be aborted
can_be_paused	True if execution can be paused
can_be_enoughed	True if execution can be *enoughed*

Figure 4: Node execution information provided by a TST Executor in form of boolean flags.

After a successful delegation the TST Executor performs its task. It is achieved by the TST Execution System invoking the following methods:

- `prepare()`
- `start()`
- `abort()`
- `{pause|continue|enough}_requested()`

Execution of an elementary task through an executor starts by invoking the `prepare()` method. Along with performing a number of predefined housekeeping actions, a check is carried out to verify whether the system is ready for execution. This method is similar to `check()` (called during the delegation phase) but it is invoked before imminent execution of the task. This is done because some time might have passed since the delegation phase and the circumstances might have changed making the platform no longer ready to perform the task. If the platform is ready, however, a call to `start()` spawns the execution thread and the actual activity begins.

The task defined by a specific TST Executor can finish in a number ways depending on the previously provided execution information. First, it can naturally run its course, for example, when the platform arrives at the desired location or completes a landing. Second, if the TST Executor allows for it, the execution can be stopped by invoking the `abort()` method. Third, for tasks which do not have a defined end condition, for example, patrolling a region or following a target, a call to `enough_requested()` finishes the execution. An additional way to influence a task's execution is to temporarily suspend it. This is possible if the `can_be_paused` flag is set to *true*. If that is the case, invoking `{pause|continue}_requested()` methods will result in suspending and resuming (respectively) the currently executed task.

Throughout the complete lifecycle a TST Executor has to provide status information, shown in Fig. 5. The combination of the boolean flags fully encodes the state of the executor at any give time. After a call to `prepare()`, the `active` flag is set to *true*. The `waiting` flag indicates that the executor is waiting for housekeeping activities to finish. The `executing` flag is set to *true* after the call to `start()`. The `finished` flag set to *true* specifies that the executor has finished. The `aborted` flag is set to *true* when the current activity has been successfully aborted. The `succeeded` flag is set to *true* if the execution finished in a nominal way. The `paused` flag is set to *true* when a successful pause was executed. The textual field `fail_reason` provides means of introspection for a human operator.

Name	Meaning
active	True if executor exists and has started a successful *prepare()*
waiting	True after start when waiting for all wait conditions to be satisfied
executing	True when all the wait conditions are met
finished	True if the executor has finished
aborted	True if the execution of the node was aborted
succeeded	True if the execution of the node was successful
paused	True if the execution of the node is paused
fail_reason	Reason for a failure

Figure 5: TST Executor status. All fields except a string fail_reason are boolean flags.

Implementing TST Executors using the methods and providing the information described above allows to transition from a platform independent task specification to a platform-specific instance of a task. The provided methods allow for specifying context in form of constraints as well as for expanding the leaf nodes into sub-nodes if necessary. Providing feedback information and executing specific platform functionalities is achieved through appropriate calls to platform's API. The following sections provide a description of such an API for VTOL platforms.

IV. Low-level control system interface

The notion of flight behavior definition and control is central to performing autonomous missions. As previously stated, independently of the complexity and the source of

a mission (e.g. a task planer, a template, pre-programmed) the result is a sequence of elementary tasks a robot should perform to fulfill the goal of a mission. These typically include actions such as take-off, flying to a number of waypoints, landing and so on. The most common way of implementing the elementary actions is through the use of control functions implementing continuous control laws. The switching between these actions is performed through transitions in a hybrid automata. It combines continuous control laws with discrete mode switching. The downside of this approach is that the transitions might be non-trivial requiring helper states in the automata to deal with discontinuities. Additionally, the number of the existing control functions for performing specific actions or maneuvers is usually fixed. Adding new functionalities requires either restructuring and re-parametrizing of existing methods or implementing completely new ones which is usually time consuming.

For these reasons a preferred solution is to take advantage of an approach based on a highly parametrizable interface which interacts with the underlying control functionalities. The idea is based on configurable *flight commands* which govern *horizontal*, *vertical* and *heading* control channels (Fig. 6a). Selecting from a number of modes for each of these channels and setting their parameters to achieve the wanted behaviors is very flexible. Additionally, organizing these parametrized commands into sequentially executed list allows for specifying a wide range of flight behaviors. Furthermore, the idea employs a setpoint generation step which assures continuity and bounds of the setpoints for the underlying control functionalities. It is configured using the *flight commands* on the input side and allows for specifying constraints on accelerations, speeds and positions being generated on the output side (Fig. 6b). Moreover, it increases safety of operation by, for example, making it impossible to command a UAV to leave a designated operational area or to exceed a safe speed in the context of the mission being executed. The setpoint generation layer also alleviates the problems associated with traditional mode switching as it assures continuity of setpoint signals for all possible flight commands. Finally, the generated setpoints are used by the underlying control system scheme, for example based on cascaded PID controllers (Fig. 6c).

A detailed description of the approach has been presented in [11]. The main concepts related to the work presented here are described in the remainder of this section.

A. Flight command structure

Flight commands are a central component of the method for interfacing with an underlying control system. A *flight command* consists of three main components managing the control channels, namely *horizontal*, *vertical*, and *heading*, as well as a *miscellaneous* component dealing with aspects such as an end condition of a command.

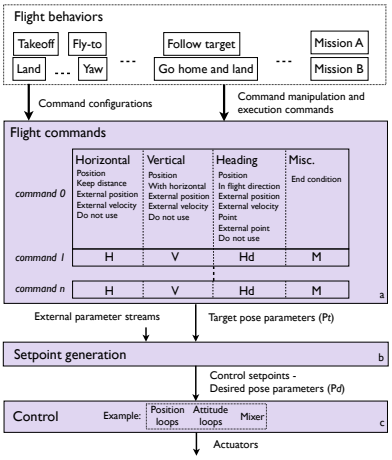


Figure 6: Low-level control system structure and interface: (a) flight commands, (b) setpoint generation and (c) control system scheme.

1) Horizontal channel: The horizontal channel governs movement of a UAV on a plane. The following modes with parameters and modifiers (see Fig. 7a) are available for the horizontal control channel:

- **Position** (H_{Pos}): Fly to an *absolute*, *relative* or *body relative* 2D position with the specified *speed* and have the specified *end speed* upon arrival. Speed can be modified when used with appropriate vertical or heading commands (see below).
- **External position** (H_{ExtPos}): Similar to H_{pos} but the parameters are provided externally as a stream of values. *Offset* parameters allow to specify a constant offset from a target position (e.g. 2m to the north of the target).
- **Keep distance**: Fly at a constant *distance* to an externally provided target position.
- **External velocity** (H_{ExtVel}): Fly with the velocities provided from an external source (e.g. a joystick).
- **RC velocity** (H_{RCVel}): Fly with the velocities provided by the RC transmitter of the backup pilot calculated onboard the platform.
- **Do not use** (H_{DNU}): Do not use the horizontal flight channel i.e. the setpoints governing this channel will remain constant.

Upon arriving at the desired position, the *end flag* (H_{end}) is set and can be used to indicate the end of the current command execution (see Subsection IV-A4). The end condition is evaluated based on the position setpoints rather than the actual values (e.g. GPS position). This results in the setpoints arriving exactly at the desired values (e.g. a waypoint).

2) Vertical channel: The following modes with parameters and modifiers (see Fig. 7b) are available for the vertical control channel:

- **Position** (V_{Pos}): Equivalent of H_{Pos} for the vertical channel i.e. fly to a specified *absolute* or *relative* altitude with specified *speed* and have *end speed* upon arrival.
- **With horizontal** ($V_{WithHor}$): Extends the H_{Pos} to a 3D case, and results in flying a straight line path to a waypoint specified including the altitude.
- **External position** (V_{ExtPos}): Equivalent of H_{ExtPos} for altitude. *Offset* allows to specify a constant vertical offset from a target (e.g. 2 m above the target).
- **External velocity** (V_{ExtVel}): Fly with the vertical velocity provided from an external source (e.g. a joystick).

Control Kernel	Parameters	Modifiers	Output
Position (H_{Pos})	north east	absolute relative body relative	end flag (H_{end})
	speed	vertical, heading	
	end speed	-	
External position (H_{ExtPos})	offset north offset east	-	end flag (H_{end})
	speed	vertical, heading	
Keep distance ($H_{KeepDist}$)	distance	-	end flag (H_{end})
	speed	vertical, heading	
External velocity (H_{ExtVel})	-	-	end flag (H_{end})
RC velocity (H_{RCVel})	-	-	-
Do not use (H_{DNU})	-	-	-

(a) Horizontal

Mode name	Parameters	Modifiers	Output
Position (V_{Pos})	altitude	absolute relative	end flag (V_{end}) speed modifier (V_{sm})
	speed	heading	
	end speed	-	
With horizontal ($V_{WithHor}$)	altitude	-	end flag (V_{end})
External position (V_{ExtPos})	offset altitude	-	end flag (V_{end}) speed modifier (V_{sm})
	speed	heading	
External velocity (V_{ExtVel})	-	-	end flag (V_{end})
RC velocity (V_{RCVel})	-	-	-
Do not use (V_{DNU})	-	-	-

(b) Vertical

Mode name	Parameters	Modifiers	Output
Position (Hd_{Pos})	heading	absolute relative stop at heading	end flag (Hd_{end}) speed modifier (Hd_{sm})
	rate	heading	
In flight direction ($Hd_{FlightDir}$)	offset	-	speed modifier (Hd_{sm})
	rate	-	
External position (Hd_{ExtPos})	offset	-	end flag (Hd_{end}) speed modifier (Hd_{sm})
	speed	heading	
External velocity (Hd_{ExtVel})	-	-	end flag (Hd_{end})
RC velocity (Hd_{RCVel})	-	-	-
Point (Hd_{Point})	north east offset rate	-	end flag (Hd_{end})
External point ($Hd_{ExtPoint}$)	offset	-	end flag (Hd_{end})
Do not use (Hd_{DNU})	-	-	-

(c) Heading

Figure 7: Control channel modes with parameters and modifiers.

- **RC velocity** (V_{RCVel}): Fly with the vertical velocity provided by the RC transmitter of the backup pilot calculated onboard the platform.
- **Do not use** (V_{DNU}): Do not use this flight channel.

The channel has two additional flags: *take-off* and *allow landing*. These are used to indicate that the commands are in fact take-off or landing maneuvers, respectively.

3) Heading channel: The following modes with parameters and modifiers (see Fig. 7c) are available for the heading control channel:

- **Position** (Hd_{Pos}): Set the heading to a specified *absolute* or *relative* value with the specified *rate*. If *stop at heading* is false, the yawing will be carried out continually with the specified *rate*. The *use sign* flag allows to force the rotation direction. If set to false, the rate sign will be ignored and the closer rotation direction will be used.
- **In flight direction** ($Hd_{FlightDir}$): Set the heading to the *offset* value relative to the flight direction (e.g. heading along the path when *offset* is 0, or fly "backwards" for *offset* of 180 degrees).
- **External position** (Hd_{ExtPos}): Set the heading to a value provided from an external source.
- **External rate** (Hd_{ExtVel}): Yaw with the rate provided from an external source (e.g. a joystick).
- **RC rate** (Hd_{RCVel}): Yaw with the rate provided by the RC transmitter of the backup pilot calculated on board.
- **Point** (Hd_{Point}): Set the heading to point towards a specified location plus an *offset*.
- **External Point** ($Hd_{ExtPoint}$): Similar to Hd_{Point} but target location point is provided externally.
- **Do not use** (Hd_{DNU}): Do not use this flight channel.

4) Command end condition: The condition allows for specifying when the current command should be considered finished. The end condition consists of a number of elements with parameters summarized in the table in Fig. 8. The three elements are evaluated as a conjunction if more than one type is used at a time.

The first type of the end condition is related to the three command channels (Fig. 8 - Channel). Depending on their configuration, H_{end}, V_{end}, Hd_{end} output flags are set to true when appropriate. This happens, for example, when a waypoint position is reached, or the desired altitude or heading are achieved. Then, these flags are evaluated if E_H, E_V, E_{Hd} flags (respectively) of the end condition are specified (or if any of the channel flags are true for E_{Any}).

The second type (Fig. 8 - User confirmation) of the end condition can be specified to allow finishing a command only if an explicit user confirmation is provided through an external signal (E_{User}). The confirmation can be performed at any point of execution of the current command.

Name	Parameters	
Channel	horizontal (E_H)	any (E_{Any})
	vertical (E_V)	
	heading (E_{Hd})	
User confirmation	- (E_{User})	
Wait	time (E_{Wait})	

Figure 8: Flight command end condition structure with parameters.

The third type (Fig. 8 - Wait) allows for specifying a timeout which triggers ending of a command (E_{Wait}). If a flight command consists only of a *wait* end condition it effectively becomes a pause.

B. Sequencing of flight commands

A single flight command composed of a selection of modes and their parameters allows for defining already expressive but rather simple flight behaviors. For this reason the commands are organized and executed as sequences. Thanks to this functionality, the range of the achievable behaviors is extensive. It can be a single command as, for example, flying to a waypoint, or a longer sequence which covers a complete mission from take-off to landing. Implementing TST Executors using the sequences of the flight commands takes advantage of this fact. Examples are provided in Section VI.

C. External data

A number of flight command modes allow the system to be configured to accept external streams of data influencing the flight behavior. The purpose of these modes is to allow more flexibility when it comes to changing flight behavior in a timely manner, for example, in a closed loop control fashion. A typical use case is a joystick commanding velocities to the platform. Only one flight command is used in such a case along with a stream of the desired velocities

received continuously and controlling one or more channels (see examples in Section VII).

Beside the parameters required for the particular modes, which are the same as for their non-external counterparts (cf. tables in Fig. 7), other kinds of data are also expected with the external data streams. For velocity commands, a validity time of the data is provided. After this time, if no new data is received, the system sets the desired values to zeroes. This prevents a platform from continuing to fly if communication is lost with the entity providing the data.

D. Setpoint generation and control

The interplay of the setpoint generation step with the underlying control system plays a major role in the proposed system. First, it allows for arbitrary and seamless change or adjustment of commands being executed avoiding the discontinuities associated with mode switching. For example, during flying to a waypoint: it can be switched to another waypoint; the speed of flight can be adjusted; an operator can take full or partial control (e.g. adjust altitude, heading), or it can directly transition into a landing, and so on. Second, it assures that the generated setpoints are compatible with the platform's properties and its control system. Additionally it allows for applying constraints of a mission being executed by, for example, limiting the maximum allowed velocities.

A schematic of the process of generating control signals based on configurations of flight commands is presented in Fig. 9.

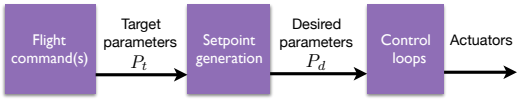


Figure 9: An overview of the process of calculating control signals based on flight commands.

The configuration of a flight command being executed produces a set of target parameters P_t which includes target positions, speeds, rates etc. Depending on the used modes of the three channels, the set of target parameters varies. The role of the setpoint generation module is to produce trajectories of desired parameters for positions, velocities, and accelerations: $P_d = [\vec{p}_d, \vec{v}_d, \vec{a}_d]$ based on P_t. This is achieved using the constant acceleration model and taking into account the constraints in form of allowed maximum values $P_{env} = [\vec{p}_{env}, \vec{v}_{env}, \vec{a}_{env}]$.

The final element is a low-level control system capable of executing trajectories of desired parameters P_d. One example of such a system is a cascade of PID loops as presented in [11]. It consists of a set of outer control loops which control position and produce input to inner loops which control attitude angles. The output is then fed into a mixer which produces final signals for servo motors or speed controllers. This approach is sufficient for most typical UAV missions for which functionalities such as aggressive or aerobatic maneuvers are not needed.

V. Intermediate interface

The low-level interface and control system described in the previous section are computationally lightweight and suitable for implementing in microcontroller class processors which are typically used for this kind of application. This allows for achieving robust and realtime performance as there is no operating system overhead. Typically microcontroller code (i.e. firmware) is executed as a single thread with statically scheduled tasks. However, to allow for integration with higher level middleware as, for example, the Robot Operating System (ROS), functionalities such as TCP/IP connectivity are required.

For this reason, the low-level control functionalities described in the previous section are wrapped into an intermediate interface called Quad API (QAPI). It allows for preserving the realtime properties of execution of tasks and, at the same time, easy integration with ROS-based system components. A general schematic of the modules involved in implementing the QAPI is presented in Fig. 10. Some aspects of the interface, such as the streaming of the external data to the system as well as requesting and receiving telemetry data, are omitted for clarity.

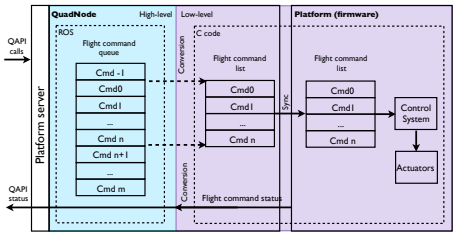


Figure 10: Integration of the QAPI with the low-level control system.

There are two main components involved in implementing the QAPI. First, a ROS-based QuadNode which implements service calls of the QAPI as well as advertises data topics which are derived from the platform's telemetry data. Second, the platform's firmware which implements the low-level control system and interfaces with the hardware. The two components interact with each other to allow accessing the flight control functionalities through a ROS interface which makes it straightforward to integrate with the TST Execution System.

A ROS message equivalent of flight commands are put on a queue (see left side of Fig. 10) to define the wanted flight behaviors as described in Section IV-B. This allows for creating behaviors consisting of virtually unlimited number of flight commands. Due to hardware memory limitations typical of microcontrollers, only a sub-queue is actually synchronized with the platform's firmware at any given time.

This is done by first converting a ROS representation of a flight command to its C-code equivalent and transferring it to the microcontroller. During the execution of a flight behavior, status information is transferred back in form of telemetry data to allow for advancing of the command queue and thus updating the flight command list on the platform.

The QAPI consists of the following methods:

- `add_command(command)`
- `run()`
- `stop()`
- `pause_direct(behaviour_id)`
- `pause_after_current(behaviour_id)`
- `reset()`
- `user_confirm()`

There are two methods which allow for manipulating the flight command queue. First is the `add_command(command)` which allows for appending commands represented as ROS messages to the queue. The second, `reset()`, allows for clearing the contents of the queue. If any command was being executed by a platform it is immediately stopped and the platform is brought to a hover state.

The execution of the command queue is started using the `run()` method. This can be done at any time, that is after a desired sequence of commands is put on the queue or even before. In that case the execution is started as soon as any command is placed on the queue. The execution can be stopped using the `stop()` method and it means not advancing to the next command and the currently executed command will be immediately stopped if allowed. The command being executed can be paused directly (`pause_direct(behaviour_id)`) or after the current command finishes (`pause_after_current(behaviour_id)`). Both kinds of pauses can be invoked with an optional `behaviour_id`, to be executed during the pause. A behavior in this context is a predefined sequence of one or more flight commands which is used for a number of typical behaviors such as joystick control (ExtVel). After this predefined behaviour is finished, the execution resumes with the next command on the list. The final method of the QAPI is `user_confirm()`. It is used to provide an external user confirmation as part of an end condition of a flight command (cf. end condition in Section IV-A4).

A typical flow of defining flight behaviors and executing them using the QAPI is the following. First, one or more flight commands are added to the queue. Then the execution is started. If needed the execution can be paused or a user confirmation is sent. Depending on a command being executed, a stream can be provided to the system as external data using an appropriate ROS topic. Examples of this process are presented in the following section where a number of TST Executors are described and the corresponding flight commands presented.

The following information is provided by the QAPI to allow for determining the status of the execution of flight functionalities:

- `queue_system_status`: Running, Overridden.
- `queue_status`: Stopped, Running, Waiting for behavior and stopping, Waiting for behavior and continuing.
- `current_command_index`: Current index of the command being executed.
- `current_command_status`: Unknown, Stopped, Running, Paused.
- `confirmation_required_status`: The command being executed requires a user confirmation to finish.
- `limit_reached`: Position limit reached.
- `target_position_unreachable`: Externally provided position is outside the allowed region.
- `is_synced`: True if the sub-queue is synchronized.

The onboard system provides a number of status variables sent as telemetry data. Based on the information the complete state of the onboard execution is known at any time. The information can be used both as feedback of the invoked methods as well as status information for the operator. The `queue_system_status` informs whether the onboard system is ready to accept the QAPI commands, or it is overridden by the backup pilot. This is part of the safety system which allows the backup pilot to override the commands being executed. The `queue_status` is updated in response to `run()` and `stop()` methods. The `current_command_index` specifies which command is currently executed, or will start executing if `run()` is invoked. The `current_command_status` provides information about the current command. The `confirmation_required_status` indicates that the current command is expecting a user confirmation in order to finish. The `limit_reached` flag informs whether the platform has reached a position limit. This provides feedback for commands for which the resulting position cannot be predicted in advance (e.g. velocity based control). The `is_synced` flag informs whether the firmware command list reflects the current ROS queue.

The ROS-based QuadNode which provides the QAPI functionalities can reside either onboard the platform on a computer separate from the microcontroller, or can be executed an a ground station computer. The communication between the ROS node and the firmware is physically realized though an RS232 connection in the case of the LinkQuad platform. It can be wired if the two components are executed onboard the platform, or wireless, if they are executed on remote computers.

VI. Examples of TST Executor implementations

Previous sections described the mechanisms involved in implementing TST Executors as well as a low-level control

system and its interface which allows for achieving different flight behaviors through parametrization of *flight commands*. This section presents how a number of executors are implemented using these functionalities.

It shows how platform independent definition of tasks are instantiated using a platform-specific API. Thanks to the expressiveness of the flight command-based interface presented in Sections IV and V, it is straightforward to map typical tasks to appropriate flight commands to achieve the desired flight behavior. Moreover, it is easy to achieve behaviors which traditionally would require additional control functionalities or deep restructuring of the existing ones.

Along with the general information about executor implementations in form of flight command configurations, information about the specific aspects of the platform used for experimental validation (see Section VII), the LinkQuad, are additionally provided where applicable.

A. Take-off

The *take-off* executor is responsible for making the platform airborne and ready to execute subsequent tasks. This executor does not have any input parameters. The typical take-off parameter, the final altitude, is not used in the *take-off* executor. The reason is that the executor can be used by different types of platforms with different requirements, for example VTOL as well as fixed-wing configurations. Therefore this and other parameters are not specified and are used internally as needed by the specific executor implementation.

The configuration of the low-level flight command which implements the *take-off* task is presented in Fig. 11. Horizontal and heading channels of the flight command are not used when configuring the take-off (H_{DNU} and Hd_{DNU}). The vertical channel uses the position mode (V_{Pos}) and the *take-off* flag is set to *true*. The altitude to which the take-off is performed depends on whether it takes place indoors or outdoors and is typically set to 2 and 5 meters, respectively, for the LinkQuad platform. The vertical channel end condition flag (E_V) is used to finish the task.

Horizontal	Vertical	Heading	End condition	Flags
DNU	Pos	DNU	E_V	Takeoff

Figure 11: *Take-off* executor flight command configuration.

The *take-off* task implemented on the LinkQuad platform cannot be paused, aborted, or stopped after it has been initiated which is reflected by setting all the execution information flags to *false*.

B. Fly-to

The *fly-to* executor defines one of the most common elementary tasks and is used to bring a platform to a specified location. The parameters and their meanings are summarized in Fig. 12a.

Name	ROS Type	Meaning
position	geopoint	Geographical location of the target position.
commanded-speed	float64	Wanted speed in m/s.
speed	string	Qualitative specification:"slow","standard","fast".
follow_ground_flag	bool	Ground relative flight.
follow_ground_altitude	float64	Distance to the ground if follow_ground_flag is true.
fly_straight_line_flag	bool	Enforce straight line flight to the target position.

(a) Executor input parameters.

Alternatives	Horizontal	Vertical	Heading	End condition	Flags
1	Pos	Pos	DNU	E_H, E_V	-
2	Pos	WithHor	DNU	E_H, E_V	-

(b) Flight command configuration alternatives.

Figure 12: *Fly-to* executor parameters and configuration of the flight commands.

The target *position* is specified as a *geopoint* (longitude, latitude, and elevation above WGS84 ellipsoid). The desired speed of the flight can be specified in two ways: as *slow*, *standard*, or *fast* using the *speed* parameter, or as a specific value using the *commanded-speed* parameters expressed in m/s. The *fly_straight_line_flag* is used to specify that the flight should be carried out along a straight line if set to *true*. Otherwise, the shape of the path to fly along is not important from the perspective of the mission. The remaining two parameters specify how the altitude of the flight should be handled. If *follow_ground_flag* is *true*, the flight should be carried out with the distance of *follow_ground_altitude* meters above the ground. It this case, the executor is also responsible for making sure all the facilities for detecting the altitude above ground are present (sensors and/or algorithms).

Two alternatives of the flight commands used to implement the *fly-to* executor are presented in Fig. 12b. Horizontal channel of the flight command uses the position mode (H_{Pos}). The longitude and latitude of the geopoint are recalculated into a local coordinate system (i.e. *north* and *east* in Fig. 7a). The mode of the vertical channel depends on the value of the *fly_straight_line_flag* and is set to $V_{WithHor}$ or V_{Pos} (for *true* and *false*, respectively). The heading remains constant throughout the flight as the Hd_{DNU} mode is used.

The LinkQuad specific *fly-to* executor can be paused and aborted i.e. the `can_be_aborted` and `can_be_paused` fields are set to *true* in the `node_execution_info` structure.

C. Land

The role of the *land* executor is to perform a landing at a current or a specified location, depending on the values of the input parameters summarized in Fig. 13a.

If the landing maneuver is required to be performed at a specific location, the *land_at_current_position_flag* is set to *false*. In that case, the *geopoint* specifies the requested land-

Name	ROS Type	Meaning
position	geopoint	Geographical location of the landing.
land_at_current_position_flag	bool	Use *position* or current location.

(a) Executor input parameters.

Seq. no.	Horizontal	Vertical	Heading	End condition	Flags
1	Pos	WithHor	Pos	E_H, E_V, E_{Hd}	-
2	DNU	Pos	DNU	E_V	Allow landing

(b) Flight command configurations.

Figure 13: *Land* executor parameters and the flight commands.

ing location. Otherwise, the landing should be performed at, or close to, the current location.

In the former case, the *land* executor uses both commands presented in the table in Fig. 13b. The first one, is used to bring the platform to the required geopoint location (similarly to a *fly-to*). The second command is configured as follows: the horizontal and heading channels are not used (H_{DNU} and Hd_{DNU}), and the vertical channel is configured to use the position mode (V_{Pos}). Additionally, the *Allow_landing* flag is used, to enable the detection of touchdown as described in [11].

D. Wait

The role of the *wait* elementary task is to provide means to specify a timed period of inactivity of a platform. The amount of time to wait is provided as a task input parameter. The implementation of the task using the low-level flight commands interface is straightforward. All three channels are configured to be inactive (H_{DNU}, V_{DNU}, and Hd_{DNU}) and the end condition uses E_{Wait} parameter to specify the amount of time for the wait.

E. Scan-ground

The purpose of the *scan-ground* executor is to perform a flight following a scanning pattern. This is a generic functionality required in various missions, for example mapping, searching for entities of interest, end so on. Depending on the requirements of the mission different sensors can be used, for example, a camera, a laser range finder or an ARTVA (avalanche beacon). The input parameters and the configuration of the flight commands are presented in tables in Fig. 14a.

The area to be covered during the execution of the *scan-ground* task can be specified in two ways. The first is to provide coordinates of a polygon using the *area* parameter and the sequence of waypoints which guarantees the area coverage can then be calculated using the algorithm presented in [12]. The second way is to set the *waypoints_scan_flag* to true and provide the list of *waypoints* directly. In that case the following additional

Name	ROS Type	Meaning
area	geopoints	Specification of the area to scan.
waypoints_scan_flag*	bool	If true scan flying the waypoints instead of area scan.
*waypoints	geopoints	A list of waypoints.
*loiter_mode	string	Behavior at a waypoint.
*segment_flag	bool	Interpret the waypoints as a sequence of straight line segments.
*any_order_flag	bool	If true fly segments or waypoints in any order.
speed	string	"fast", "standard", "slow", unspecified.
altitude	string	"high", "standard", "low", unspecified.
follow_ground_flag	bool	Terrain following flag.
follow_ground_altitude	float64	Terrain relative altitude.
repeat_flag	bool	If this is true we are in patrol mode and repeat the scan of the area.
sensor_type	string	"artva", "laser", "camera".
data_uuid	string	Used to identify the result of the scan.
generate_scan_specs_flag	bool	If true generate scan specs and put in queue.

(a) Executor input parameters.

Alternatives	Horizontal	Vertical	Heading	End condition	Flags
1	Pos	WithHor	DNU	E_H, E_V	-
2	Pos	ExtPos	DNU	E_H	-

(b) Flight command configuration alternatives.

Figure 14: *Scan-ground* executor parameters and the flight commands.

parameters can be used. The behavior at a waypoint can be specified using the *loiter_mode*. The *segment_flag* specifies that the waypoints are describing segments, that is the flight between these segments can be performed in an arbitrary way. The *any_order_flag* allows to specify that the order of scanning segments does not have to be enforced. The *follow_ground_flag* and *follow_ground_altitude* have the same meaning as for the *fly-to* executor. The *speed* and *altitude* parameters describe the respective values qualitatively, it is up to the executor to assign appropriate platform-specific values. The *repeat_flag* allows to specify that the scanning should be done continuously. This way it is possible to execute the mission of a patrolling type. The *sensor_type* parameter allows for specifying which sensor should be used during the execution of the task. The *data_uuid* parameter specifies that the data collected during the execution of this task should be labeled with this unique identifier. The *generate_scan_spec_flag* specifies that the executor should generate a sub-area of interest to be subsequently scanned.

Two alternatives of flight commands used to implement the *scan-ground* executor are presented in Fig. 14b. The horizontal channel of both flight commands uses the position mode (H_{Pos}). The parameters of positions to fly to are re-calculated to the local coordinate system based on the list of the provided waypoints or on the output of the scan pattern generation algorithm. The vertical channel uses either $V_{WithHor}$ or V_{ExtPos} modes. The latter one is used to implement ground relative flight when *follow_ground_flag* is set to *true*. The externally provided altitude to fly at is calculated taking into account readings from a ground proximity sensor (e.g. a laser range finder).

The LinkQuad implementation of the *scan_ground* executor has all three execution info flags: `can_be_aborted`, `can_be_paused`, and `can_be_enoughed` set to *true*.

F. External-velocity

The purpose of the *external-velocity* executor is to allow for flight with velocities (including yaw rate) provided from an external source (external from the perspective of the low-level control system and provided as a ROS topic). A typical example of the usage of this executor is to enable an operator to control a platform using a joystick. It can also be used for any other application where this mode of operation is needed (e.g. visual servoing).

All three channels use the external velocity (ExtVel) modes. The command is finished by the user confirmation end condition (End_{User}). Since this type of a command has no inherent end, the finish condition has to be provided by the user.

VII. Experimental validation

The system presented in this paper has been fully implemented and used in numerous autonomous missions. This section presents results of one example mission. Its structure is of a generic *scan and search* type and is similar to the one presented in [13]. The experimental setup is as follows. The ground operator selects an area to scan for salient points (e.g. a missing person). This area is provided as an input to the *scan-ground* executor. A scanning pattern in form of a sequence of waypoints is generated. During the execution of the scan, the operator monitors the live video from the platform. Upon noticing a potential person, the operator can *brake-in* and be given direct control of the platform (velocity control using a joystick). Depending on the result of the closer investigation, the operator can *resume* the execution, or indicate that *enough* of the mission has been performed and thus skipping the remaining waypoints of the scan because the missing person has been found. In the latter case, the execution of the mission continues with a flight to a predefined location, and finishes with a landing. Fig. 15 presents the Task Specification Tree for the experimental mission. It consists of a sequence of four tasks: take-off, scanning of the area, flying to a predefined location, and performing a landing.

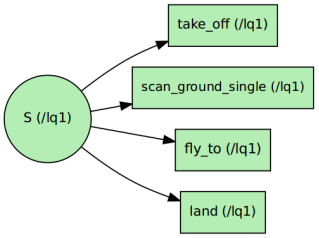
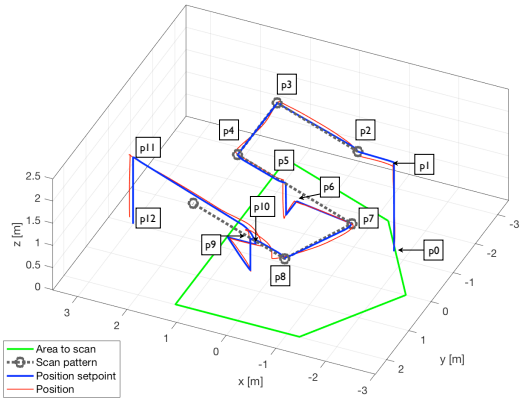


Figure 15: Task Specification Tree of the experimental mission.

The example mission was performed using the LinkQuad platform (80cm propeller tip-to-tip size) in an indoor motion capture arena[2]. The state estimate was provided externally to the onboard system as a substitute for a GPS, a compass, and a pressure sensor used for state estimation for outdoor flights. Beside this aspect, the complete system is used both in indoor and outdoor flights in an identical manner. The platform and its architecture which allows for such mode of operation is presented in [14].



(a) Position and its setpoints during the mission.

TST Executor	Segment endpoints	Horizontal	Vertical	Heading	End condition	Flags
take-off	p0-p1	DNU	Pos	DNU	E_V	Takeoff
scan-ground	p1-p5	Pos	Pos	Pos	E_H, E_V, E_{Hd}	-
scan-ground (paused)	p5-p6	ExtVel	ExtVel	ExtVel	E_{User}	-
scan-ground	p6-p9	Pos	Pos	Pos	E_H, E_V, E_{Hd}	-
scan-ground (paused/enoughed)	p9-p10	ExtVel	ExtVel	ExtVel	E_{User}	-
fly-to	p10-p11	Pos	Pos	Pos	E_H, E_V	-
land	p11-p12	DNU	Pos	DNU	E_V	Allow landing

(b) Configurations of the flight commands implementing the executors.

Figure 16: Experimental validation mission setup and results.

Fig. 16a presents the plot of the flight path of the experimental mission. The area to scan is denoted using a green line. The resulting scan pattern is depicted using a dashed gray line, and the waypoints are highlighted with circles. The position setpoints (cf. P_d in Section IV-D) are drawn using blue lines and the actual path during the flight is marked using the red line.

Fig. 16b shows the flight commands implementing the TST Executors used in the presented mission. Executors: *take-off*, *fly-to*, *land* are implemented using single flight commands while *scan-ground* consists of a sequence of flights to waypoints of the scanning pattern. Additionally, a *pause* is executed during the scan (at points p5 and p9). This results in executing a flight command which puts all the flight channels into external velocity modes. In this case the operator uses a joystick to command the platform to

[2]Arena equipped with ten T10 and six T40-S Vicon cameras covering approx. 10×10×5m volume.

go closer to the point of interest. The operator continues the execution of the scan at p6. After the second pause at p9, however, *enough* action is issued (at point p10) which finishes execution of the scan without reaching the last waypoint. The mission continues with the next executor at p10, and a flight to p11 is performed. Finally, the mission finishes with a landing.

Despite exchanging flight commands during the flight, the setpoints remained continuous from takeoff to landing. All flight behaviors were achieved by appropriately parametrizing the modes of flight commands. The maximum absolute control error measured for the horizontal channel was: 39.1cm with root-mean-square (RMS) of 14.1cm. For the vertical channel (excluding takeoff) the values were: 19.2cm and 5.4cm, respectively. This shows that the underlying control system using cascaded PID loops provides sufficient control accuracy for this kind of mission.

VIII. Conclusion

We have presented an approach to connecting mission-level task definitions with low-level control functionalities. This has been achieved taking into account architectural requirements of missions involving multiple heterogenous UAV platforms. We have described how platform-specific implementations of parametrized generic tasks can be realized using a flexible low-level control system interface. This has been done in the context of a previously described robotics architecture. We have shown that by using a flight-command based interface and control system at an appropriate abstraction levels it is straightforward to implement platform-specific task executors. This has allowed for implementing a wide range of common tasks only by appropriate parametrization of the control interface. It did not require any development of new flight control functions. The presented system has already been used in numerous missions and one example mission has been presented and described.

Acknowledgment

This work is partially supported by the Swedish Research Council (VR) Linnaeus Center CADICS, the ELLIIT network organization for Information and Communication Technology, and the Swedish Foundation for Strategic Research (SymbiKcloud Project). The authors would like to acknowledge the software development support of Tommy Persson.

References

[1] P. Doherty, F. Heintz, and J. Kvarnström, "High-level mission specification and planning for collaborative unmanned aircraft systems using delegation," *Unmanned Systems*, vol. 01, no. 01, pp. 75–119, 2013. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S2301385013500052

[2] E. Gat, R. P. Bonnasso, R. Murphy, and A. Press, "On three-layer architectures," in *Artificial Intelligence and Mobile Robots*. AAAI Press, 1997, pp. 195–210.

[3] J. D. Boskovic, R. Prasanth, and R. K. Mehra, "A multi-layer autonomous intelligent control architecture for unmanned aerial vehicles," *Journal of Aerospace Computing, Information, and Communication*, vol. 1, no. 12, pp. 605–628, 2004.

[4] F. Adolf and F. Andert, *Onboard mission management for a VTOL UAV using sequence and supervisory control*. INTECH Open Access Publisher, 2010.

[5] P. Doherty, J. Kvarnström, M. Wzorek, P. Rudol, F. Heintz, and G. Conte, "HDRC3 - a distributed hybrid deliberative/reactive architecture for unmanned aircraft systems," in *Handbook of Unmanned Aerial Vehicles :*, 2014, pp. 849–952.

[6] S. hyun Lee, S. H. Kang, and Y. Kim, "Trajectory tracking control of quadrotor uav," in *Control, Automation and Systems (ICCAS), 2011 11th International Conference on*, Oct 2011, pp. 281–285.

[7] D. Cabecinhas, R. Naldi, L. Marconi, C. Silvestre, and R. Cunha, "Robust take-off for a quadrotor vehicle," *Robotics, IEEE Transactions on*, vol. 28, no. 3, pp. 734–742, June 2012.

[8] D. Cabecinhas, R. Cunha, and C. Silvestre, "A robust landing and sliding maneuver controller for a quadrotor vehicle on a sloped incline," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, May 2014, pp. 523–528.

[9] M. Hehn and R. D'Andrea, "A frequency domain iterative learning algorithm for high-performance, periodic quadrocopter maneuvers," *Mechatronics*, vol. 24, no. 8, pp. 954–965, Dec. 2014.

[10] O. Andersson, M. Wzorek, and P. Doherty, "Deep learning quadcopter control via risk-aware active learning," in *Thirty-First AAAI Conference on Artificial Intelligence (AAAI), 2017, San Francisco, February 4?9.*, 2017.

[11] P. Rudol and P. Doherty, "Bridging the mission-control gap: A flight command layer for mediating flight behaviours and continuous control," in *2016 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, 2016.

[12] C. Berger, M. Wzorek, J. Kvarnström, G. Conte, P. Doherty, and A. Eriksson, "Area coverage with heterogeneous uavs using scan patterns," in *2016 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. IEEE Robotics and Automation Society, 2016.

[13] P. Doherty, J. Kvarnström, P. Rudol, M. Wzorek, G. Conte, C. Berger, T. Hinzmann, and T. Stastny, "A Collaborative Framework for 3D Mapping using Unmanned Aerial Vehicles," in *PRIMA 2016: Principles and Practice of Multi-Agent Systems*, ser. Lecture Notes in Computer Science, vol. 9862. Springer Publishing Company, 2016, pp. 110–130.

[14] M. Wzorek, P. Rudol, G. Conte, and P. Doherty, "LinkBoard: Advanced flight control system for micro unmanned aerial vehicles," in *IEEE International Conference on Control and Robotics Engineering (ICCRE)*, 2017.