An Execution Engine for Aerial Robot Mission Plans

Martin Molina
Department of Artificial Intelligence
Technical University of Madrid, Spain

Technical Report
June 29, 2017

Abstract

The goal of the work presented in this paper is to develop a practical solution for mission plan execution to simplify the way in which operators configure the missions of robots. This work has been done to promote a more extensive use of the software framework for aerial robotics Aerostack. We have designed a computer system called execution engine that includes technical solutions from general robotics and artificial intelligence. The system follows a behavior-based approach and a symbolic representation of beliefs. The execution engine has been designed to be part of Aerostack but it can also work independently, so that it can be reused for building other type of robot architectures. This paper has been written as a specification and software design to be used as a guide for software implementation of the execution engine.

TABLE OF CONTENTS

1. Introduction

Aerostack is a software framework that helps developers design the control architecture of an aerial robot, integrating multiple heterogeneous computational solutions for autonomous behavior (e.g., computer vision algorithms, motion controllers, planning algorithms, etc.). Aerostack provides a powerful library of software components for robotics and a combination scheme for building the final architecture. Aerostack has demonstrated to be an effective tool for building different types of aerial systems in complex and dynamic environments [Sanchez-Lopez et al., 2016; 2017]. It has proved to be a useful research platform to support flight experiments that evaluate new approaches in aerial robotics (e.g., [Suárez-Fernández et al., 2016; Molina et al., 2017]). Aerostack was created to be available for different communities of developers and it is currently an active open-source project with periodic software releases (www.aerostack.org).

Programmers who are familiar with Aerostack (e.g., programmers that belong to the development team of Aerostack or experienced programmers in aerial robotics) may use of the Aerostack library of components for rapid construction of a control architecture for an aerial platform. However, this is not easy for other potential users of Aerostack. The main problem is that the current version of Aerostack assumes that the programmer knows many low-level technical details, and it is not protected against certain errors. Therefore, Aerostack can be difficult to manage and error-prone for general users.

In this paper we present results of our recent work to promote a more extensive use of Aerostack. In principle, to achieve this goal, Aerostack admits different improvements (e.g., graphical user interfaces with additional communication methods, etc.). Our emphasis in the work presented in this paper is on issues related to the practical execution of mission plans that can simplify the operation with robots.

As a result of this work, we have designed a new software system, that we call *execution engine,* that provides the robot with effective execution capabilities. This system incorporates a number of technical solutions used in robotics and artificial intelligence. The resulting system creates a new logical interface with Aerostack that accepts instructions from the operator in a simpler way, encapsulating in more robust components the libraries of basic components of Aerostack.

This paper has been written as a specification and general design to be used as a guide for the software implementation of the execution engine with the corresponding programming languages. The paper describes engineering details of our efforts to improve Aerostack, showing the reasons that support our design decisions, which can be useful for developers to better understand the architecture of this framework. The paper presents the execution engine, that was created to be part of Aerostack, but it has been designed to be general and, therefore, reusable for other robot architectures.

The remainder of the paper presents in detail the results of this work. Section 2 identifies the execution needs for a more effective operation, following ideas from the state of the art in robotics and artificial intelligence. Section 3 shows an overview of our execution engine. Section 4 and 5 explain the two components of the execution engine. Section 6 describes how the execution engine can be used by mission plan interpreters.

2. Execution needs for effective operation

The goal of this section is to place the objectives of our work in the context of robotic systems, to identify and justify the execution capabilities that are needed to simplify the user operation with aerial robots during the specification and the execution of mission plans.

2.1. Human-robot interaction based on supervisory control

In the work presented in this paper, we assume that the human-robot interaction is based on supervisory control [Sheridan, 1992] (Figure 1). According to this form of control, the relation between operator and robot follows a hierarchical authority. A human operator acts as a supervisor and the robot as a subordinate. The operator gives directives to delegate mission tasks to the robot. The robot understands and translates the directives into detailed actions by the robot. The robot collects detailed information about results and presents it to the supervisor. In supervisory control, the human operator programs the robot and receives information from the robot that itself closes an autonomous control loop through effectors and sensors to the environment.

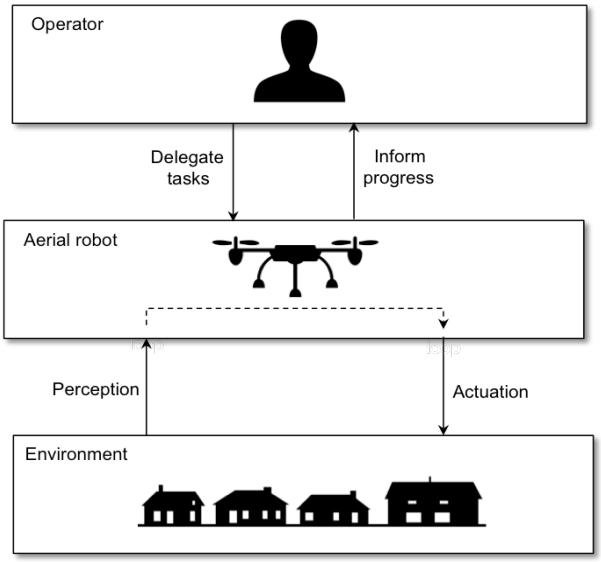

Figure 1: Human-robot interaction based on supervisory control.

For example, the operator may ask the aerial robot to perform an inspection mission, specifying the area to cover and the exploration strategy. During the development of the mission, the operator observes the robot behavior and the robot sends messages to inform about the mission execution progress. This observation is useful to confirm that the mission is developed as expected. The operator can interrupt the mission under certain circumstances (for example, to avoid wrong behaviors in unexpected situations).

One of the design decisions in this human-robot interaction is to select an appropriate *language* to be used between operator and robot with the adequate expressivity and level of abstraction. Operators use this language to describe formally the goals that the robot must achieve during the mission (including the specific criteria that uses the robot to determine the sequence of actions to achieve such goals). In robotics, different languages have been proposed to specify mission plans following a variety of representations:

- Many of languages for aerial robotics use lists of GPS waypoints with associated actions or commands. For instance, MP – Mission Planner uses navigation commands to travel to waypoints, do commands to execute specific actions (e.g., taking pictures), and condition commands that control when other commands are able to run.

- A popular approach also is using finite state machines (FSMs). This representation has been used, for example, in languages such as the Behavior Language [Brooks 1990], the Colbert language [Konolige, 1997], or MissionLab [MacKenzie 1997].

- Other approaches have proposed modular and hierarchical representations such as hierarchical finite state machines (HFSM). For example XABSL (Extensible Agent Behavior Specification Language) [Loetzsch et al 2006] [Risler, 2009], State Control Library and Behavior Control Framework5 in NimboRo-OP [Allgeuer, Behnke, 2013].

- Another hierarchical approach common in robotics is the task-based representation. In robotics, this representation has been used in ground robot control [Simmons, Apfelbaum, 1998; Nicolescu, Matarić, 2002], underwater vehicles [Roberts et al., 2003; Ridao et al., 2005] or for multi-agent UAV systems [Doherty et al 2010].

- The representation with behavior trees is another approach that uses a hierarchical representation. Behavior trees were proposed in the computer gaming industry. In robotics, behavior trees have been used recently [Marzinotto et al 2014] [Colledanchise, Ogren, 2014] and, specifically for UAVs [Ögren 2012] [Klöckner, 2013], such as the Modelica library (not free available) for UAV [Klöckner, et al. 2014].

Hierarchical representations are popular solutions for complex and adaptive missions. For example, in Aerostack we can use the TML language that follows a hierarchical approach together with reactive planning [Molina et al., 2015]. However, our emphasis in this work is not on a particular specification language for mission plans. In the work presented in this paper, we are interested on issues related to the effective execution of mission plans and the appropriate interaction with the operator during the mission execution.

2.2. Practical needs of execution systems

To execute mission plans written by human users, it is necessary to have effective interpreters that transform automatically the instructions described in the operator language into detailed commands for the robot's controllers. Autonomous robots have particular execution needs due to, for example, the mission may be executed in unpredictable environments (that makes, for example, that a planned action may be unfeasible) or the aerial robot has limited physical resources (e.g., the battery charge) that must be used efficiently.

Robot architectures with high degree of autonomy usually have an executive system in charge of these issues. The executive layer in robot architectures can be understood as the interface between the numerical behavioral control and the symbolic planning layer [Kortenkamp et al., 2016]. The execution system takes a plan that assumes a certain level of certainty and expected outcomes and makes executive decisions to guarantee the correct execution of the plan in an uncertain and dynamic environment [Verma et al., 2005; Murphy, 2000].

The practical experience in the development of robot systems shows that this part of the architecture is a critical component with a high impact on safety and quality of the communication with the operator. It is important to design and validate carefully the executive

system of a robotic system. In a robot engineering context, where productivity is essential, it is important to have methods and software tools that reduce the effort of building such executive systems

Ideally, the goal of an execution system is to achieve robust effective performance of an autonomous robots with the minimum programming effort to specify the mission. We analyze here two basic goals to consider in the development of an effective execution system that can facilitate the human operation with robots: (1) *simplicity*, plans written by operators should be described as simple as possible, and (2) *robustness*, robots should be able to react correctly to contingencies that happen in an unpredictable environment. In the following, we explain how these goals can be achieved by using certain solutions from general robotics and artificial intelligence.

2.2.1. *Simplicity*

To achieve simplicity, we can use the concept of *behavior* that provides abstraction, hiding low level technical details and complexity. A behavior is a natural notion that is familiar for general users (in simple terms, a behavior is is anything the robot is able to do: take off, move forward, land, etc.).

The concept of behavior has been traditionally used in the literature of robotics for example, in subsumption architectures [Brooks, 1986]. A behavior encapsulates a set of perception algorithms and actuation controllers to generate a particular pattern perception-actuation. According to Robin Murphy, a behavior is a direct mapping of sensory inputs to a pattern of motor actions that is used to achieve a task [Murphy, 2000]. A behavior can be understood as a control law that clusters a set of constraints in order to achieve and maintain a goal [Mataric, 1994]. According to François Michaud and Monica Nicolescu, each behavior receives inputs from sensors and/or other behaviors in the system, and provides outputs to the robot's actuators or to other behaviors [Michaud, Nicolescu, 2016].

A mission can be specified with a coordinated set of behaviors. During the execution of the mission plan, some groups of behaviors are activated and other groups are deactivated following a detailed sequence of activations and deactivations. An important issue to consider here is how to manage conflicts of behaviors that can operate concurrently. There is a conflict between two behaviors if they try to act on the same actuator at the same time with different orders. In general, this problem is called *behavior coordination* (or behavior fusion).

Mission plans may use different types of physical devices (speakers, cameras, microphones, lights, etc.). It is important manage efficiently these devices during the mission execution due to *resource-bounded operation*. Their use is associated to the increase of resource consumption (memory space, processing time, battery charge) so it is important to stop unnecessary processes when it is possible.

Besides behaviors, mission plan descriptions should able to include abstract expressions, avoiding excessive details, which helps to be more general and therefore reusable across different environments. This generality, for example, can be achieved by using specialized planners. During the execution, planners particularize the abstract expressions of the global plan with specific details of the environment. For example, an abstract plan may express that the robot must go to the *next point to explore* (without saying the specific spatial coordinates of that point). Then, during the execution of the plan, a specialized planner automatically determines the particular coordinates of the point of the environment in which the plan is executed, according to a prefixed exploration strategy.

2.2.2. *Robustness*

A useful notion to operate in unpredictable environments is the concept of *cognizant failure*. This is a design approach that states that a system should be designed to detect failures [Gat, 1997; 1998]. Robustness can be achieved by using contingency-handling constructs that express how to react to contingencies, with procedures that recover from failures. Some languages for mission plans include this approach using specialized representations based on reactive planning or conditional sequencing [Firby 1987; Gat, Dorais, 1994] in languages such as RAPs [Firby 1989] or ESL [Gat, 1997].

Thus, to gain robustness, the robot may exhibit the ability of self-monitoring its own behavior. In a context with human-robot interaction with supervisory control, the result of this monitorization should be represented in an appropriate way to be communicated to the operator, in order to help to understand unexpected robot behaviors (e.g., using symbolic representations).

3. The execution engine

Following the ideas presented in the previous section, we have designed a software system that we call execution engine. This section describes the characteristics of such a system.

3.1. Robot functional requirements to facilitate user operation

In the discussion presented in section 2, we consider two basic goals in the development of an effective execution system, simplicity and robustness, that can be achieved using different means such as the use of behaviors or cognizant failure. This lead us to the following robot functional requirements:

- *Requirement "execute behaviors*". The robot is capable of executing simple instructions of a mission plan requested by the operator in terms of behaviors. The operator can select a behavior from a set of available behaviors for the robot. The robot may activate and cancel the execution of the behavior requested by the operator. The robot executes concurrently behaviors keeping consistency with the state of the world and with other behaviors.

- *Requirement "explain itself"*. The robot is capable of describing to the operator what it believes about the environment and about its own performance. A symbolic representation is important to establish an understandable communication with operators. The robot manages a memory of symbolic beliefs to simulate awareness of the state of the world and the internal state of the robot. Beliefs can also represent the awareness of deliberative thinking done by the robot (e.g. reasoning about the physical world for motion planning, such as path planning). In addition, beliefs also help to monitor the correct execution of behaviors. The correct execution of each behavior is periodically monitored with defensive procedures that check belief expressions and beliefs are used inform more clearly to the operator when contingencies are detected. Operators can also use beliefs when they write mission plans to describe the presence of potential contingencies and adapt the execution of the plan to the current circumstances.

3.2. The execution engine in Aerostack

Figure 2 presents a diagram of building blocks showing how the execution engine is integrated to be part of the software architecture of Aerostack. The architecture includes a lower level (in green color) with a set of processes provided by Aerostack for perception, control, planning and communication. The execution engine (in blue color) creates a robust level on top of this level to offer two execution capabilities: (1) a set of processes to execute behaviors and (2) a set of processes that manage beliefs that help the robot explain itself. In its turn, the execution engine is used by a mission plan interpreter that translates the representation used by a mission plan (e.g., task trees) into service calls of the execution engine.

Figure 2: The execution engine in the architecture of Aerostack.

3.3. Software architecture of the execution engine

The architecture of the execution engine is implemented with two systems, the behavior management system and the belief management system. Each system includes a set of processes, that are implemented using ROS (Robot Operating System). Sections 4 and 5 explain in more detail such processes. These systems belong to other super-systems of the executive layer of the Aerostack architecture: the belief management system belongs to situation awareness system and the behavior management system belongs to executive system.

Figure 3 shows how these systems are connected. The figure is a block diagram (in general, in this type of diagram each block can be a system or a process). In the figure, blocks are connected with input/output ports (rounded ports correspond to ROS services and squared ports correspond

to ROS topics). The figure uses also generic names for certain ports (in italic) that establish the connection with the rest of Aerostack. For example, the generic name for the port *perception values* represents any topic generated by the perception system of Aerostack.

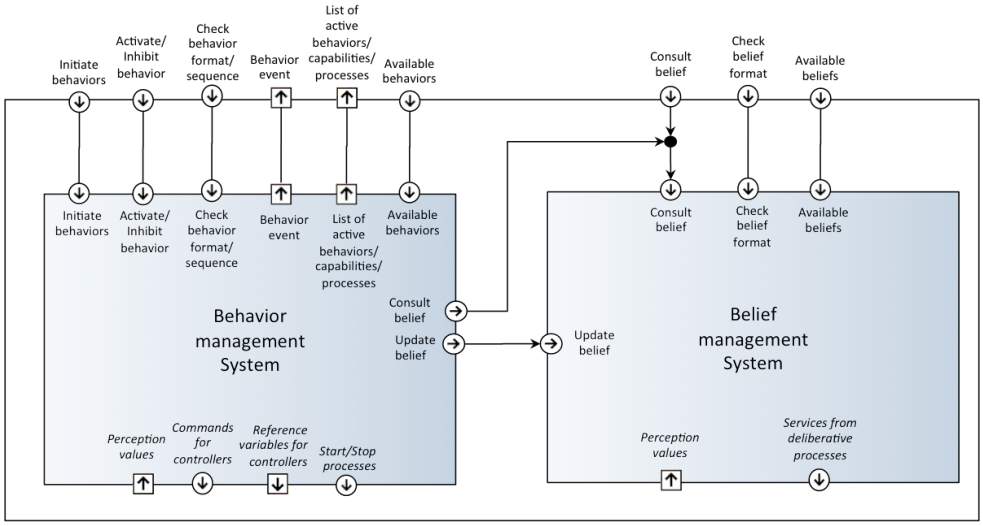

Figure 3: Architecture of the execution engine with two main systems.

3.4. Mission plan verification

One of the main functions of the execution engine is to provide robustness by verifying that the written instructions for the robot are correct. The execution engine must return adequate messages to the operator in the presence of errors to help to correct them. There are two main categories of errors, depending on when they are detected:

- *Specification errors* that are detected before the plan is executed. Specification errors correspond to mistakes done by the operator who writes the mission plan. Examples of these errors are: wrong format of the instructions (e.g., wrong names, wrong values) or goal conflicts due to unfeasible action plans that cannot be executed in any environment (e.g., the action land cannot be requested just after the same action land has been requested).

- *Execution errors* that correspond to problems detected during the execution of a mission plan. In this case, the language specification of such mission is correct but the specific characteristics of the environment where the mission is executed, and/or the particular aerial platform used, create a conflict to execute the mission.

Error type	**Text message**
WRONG_FORMAT	The behavior <x> does not exist The argument <x> of behavior <y> does not exist The format of the belief expression <x> is not correct The predicate <x> of the belief expression <y> does not exist
WRONG_VALUE	The value <x> of argument <y> for behavior <y> is not correct (... is expected)
WRONG_SEQUENCE	The execution of behavior <x> cannot be done after the execution of <y>

Table 1. Example of specification language errors.

Error type	Text message
FAILED_PRECONDITIONS	Behavior <x> cannot be executed because its preconditions are not satisfied.
TIME_OUT	The goal of behavior <x> has not been reached in the expected time (<y> seconds)
WRONG_PROGRESS	The behavior <x> does not progress correctly
ACHIEVED_GOAL	The goal of behavior <x> has already been achieved

Table 2.Example of execution errors.

According to this, we separate the verification in two parts. Before the execution, the mission plan is verified to detect specification errors. During the execution, the rest of the errors are verified. This means that, when an instruction is going to be executed, it is assumed that it has been verified previously and it does not have any specification errors.

Table 2 shows examples of conflicts to that may happen before or during the execution of behaviors. They can grouped in three categories:

- *Environment state conflict*. There is a conflict between the behavior and the current state of the environment (this behavior could be activated in other environments).
- *Robot state conflict*. There is a conflict between the behavior and the current state of the robot (this behavior could be activated if the the robot is in another state).
- *Goal conflict*. The goal is impossible to be reached by this robot independently of the state of the environment and the state of the robot (this behavior goal cannot be achieved by this robot in any environment).

Type of conflict	Example
Environment state conflict	The illumination is insufficient to activate behavior <x> There are too strong vibrations to activate behavior <x> There is an impassable barrier to complete behavior <x> There is an obstacle too close to destination point to complete behavior <x> The available space is too narrow to activate behavior <x> The ground is unstable to activate behavior <x> There are not visual markers in the field of view to activate behavior <x> The environment is not represented with enough density points to generate a path
Robot state conflict	The robot is landed so it cannot activate behavior <x> The robot does not have enough charge of battery for behavior <x> The robot is not able to estimate its position
Goal conflict	The destination is too far for behavior <x> The maximum speed is insufficient to reach the goal of behavior <x> (e.g. to reach a point in short time that is at a long distance)

Table 3. Example of conflicts to execute behaviors.

4. The behavior management system

The main goal of the behavior management system is to provide the robot with the ability of executing instructions requested by the user in terms of behaviors. This system includes four processes: behavior manager, behavior specialist, behavior process and resource manager (Figure 4). Behavior process is actually a *type* of process with several occurrences, one for each behavior as we explain below. In figure 4, processes in blue color are general for all robot systems. Processes in orange color need to be programmed to support specific behaviors.

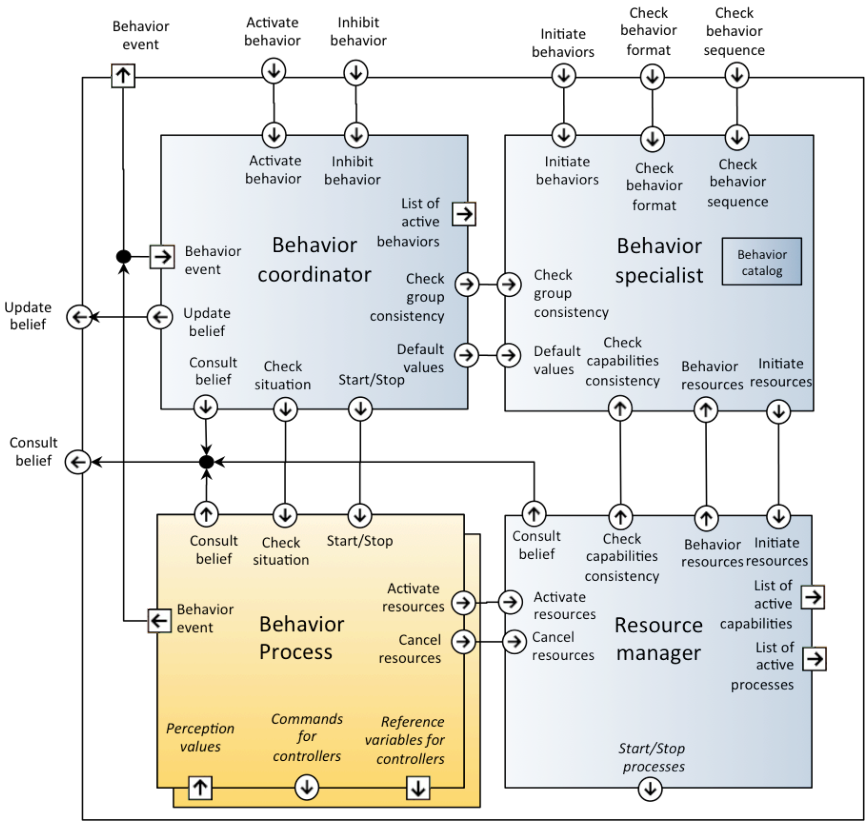

Figure 4: Architecture of the behavior management system.

4.1. The behavior process

Our architecture includes a separate process for each type of behavior. For example, there is a separate process for behavior *take off* and another one for behavior *go to point*. This separation provides modularity and robustness, since each process encapsulates the technical details for the correct execution of the behavior.

One of the significant services provided by this process is to check that the environment is consistent with the behavior execution. For example, to execute the behavior *land* the robot must be flying. Trying to execute the behavior *go to a point* where the drone is already located does not make sense. For this purpose the process provides a ROS service called *check situation* that verifies a set of preconditions. These preconditions are formulated as belief expressions (this is explain in more detail in Section 5 that describes the belief management system).

To control the execution of the behavior, this process provides the ROS services: *start* and *stop*. The possible execution states of a behavior are the same states that we use for general processes in Aerostack. Figure 5 shows the available states and transitions of Aerostack processes. Initially, once the system is running for a particular aerial robot, the process is in the state *ready to start*. The execution of the behavior is activated with the ROS service *start*, which automatically changes to the state *running*. To cancel the execution of the behavior, we use the ROS service *stop*, which changes to the state *ready to start*.

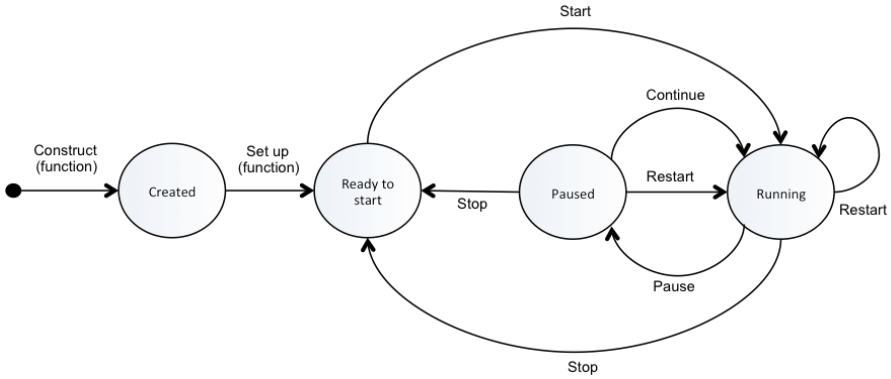

Figure 5: State-transition diagram of a process in Aerostack.

Algorithm 1: start(behavior *b*)

1. activate resources for behavior *b*
2. **if** (resources for behavior *b* cannot be activated)
3. **then** return(failure)
4. **else**
5. initiate the time measurement of behavior execution
6. extract argument values
7. connect inputs/outputs (advertise and subscribe ROS topics)
8. initiate behavior execution (read and publish ROS topics and call ROS services)
9. return(success)

Algorithm 2: run(behavior *b*)

1. *f* ← FALSE
2. **repeat**
3. **if** (wrong progress conditions are TRUE) **then** e ← WRONG_PROGRESS, *f* ← TRUE
4. **else if** (category of behavior *b* is "goal-based" or "deliberative") **then**
5. **if** (goal achieved conditions are TRUE) **then** *e* ← GOAL_ACHIEVED, *f* ← TRUE
6. **else if** (timeout of behavior *b* is TRUE) **then** *e* ← TIME_OUT, *f* ← TRUE
7. **until** (*f*)
8. publish execution result (publish event *e* in the topic "behavior event")
9. call the function stop(behavior *b*)

Algorithm 3: stop(behavior *b*)

1. write end values for inputs/outputs (read/publish topics and call services)
2. cancel resources of behavior *b*
3. disconnect input/outputs (shutdown ROS topics and services)

The detailed execution of a behavior is controlled by three functions: *start*(), *run*(), *stop*(). The function *start*() is called by the ROS service *start*. The function *run*() is automatically called when the process is in the state *running*. The function *stop*() is called at the end of the function *run*(), or inside the ROS service *stop*.

The function *start*() executes a sequence of steps that are summarized by algorithm 1. It is important to know that before this function is called, the argument values (in text format) and the timeout value (a number) are written in two fields of the behavior object. In summary the algorithm covers the following steps:

- The first step of the algorithm is to activate the resources corresponding to the behavior. As a consequence of this, the processes that support the execution of this behavior start running. If this is not possible, the behavior returns failure.
- In the second step, the measurement of execution time is initiated.
- In the third step, the process extracts the values of the arguments (if the behavior has arguments) from the text stored in the corresponding field of the behavior object.
- In the fourth step, the process establishes the input/output connections (using ROS interprocess communication methods for advertisement and subscription).
- In the fifth step, the behavior publishes initial values for certain processes if it is necessary (which in turn may require to read certain values).

Once this function has been finished, the state of the process is *running*. Note that steps three, four and five are specific for each behavior and the rest and general. This is why the first two steps are programmed in a general class (the class behavior process) and the rest are specific for each instance.

When the state of the process is *running*, the function *run*() is called automatically. This function executes the steps described by algorithm 2. This algorithm performs a loop to monitor the execution progress, reading perception values. Note that this monitoring is different in goal-based behaviors, concurrent behaviors and deliberative behaviors (Appendix A describes the categories of behaviors that we use). The loop of the algorithm can finish with success or failure and, finally, the function publishes the generated event (using the ROS topic *behavior event*) and calls the function *stop*() to stop the behavior execution, changing the state of the process to *ready to start*.

In certain execution systems behaviors generate specific events. For example, in the system designed by Firby and Slack [Firby, Slack, 1995], the behavior *watch-for-landmark* can generate the event *landmark-visible*, or the behavior *move-to-landmark* can generate the events *movement-complete*, *stuck*, or *lost-landmark*. However, in our execution engine, we consider only generic events that are common for all behaviors (if we would need to consider specific cases, they could be processed as beliefs as it explained in Section 5). For example, we consider the following generic event for recurrent behaviors: `WRONG_PROGRESS,` i.e., the behavior progress is incorrect. For goal-based behaviors, in addition to the previous event, we consider other two events: `GOAL_ACHIEVED,` i.e., the goal of the behavior has been reached, and `TIME_OUT,` i.e., the goal has not been reached in the expected time interval. In the case of deliberative behaviors, the possible events are `GOAL_ACHIEVED` or `TIME_OUT`. The result of the deliberative behavior is communicated through the ROS topic *behavior event*.

The function *stop*() executes the steps presented by algorithm 3. In the first step, the algorithm writes final values for certain processes (if necessary). The second step deactivates the resources of the behavior by stopping the running processes that support the behavior. Finally, step three disconnects inputs/outputs. The execution of this function changes automatically the process to the state *ready to start*. The function *stop*() can be also called by the ROS service *stop*. When this service is called, this means that the behavior does not stop by itself, but the operator interrupts

the execution (e.g., to inhibit a recurrent behavior) or the behavior coordinator forces the stop (e.g., because its execution not compatible with a requested behavior). Thus, the ROS service *stop* performs two actions: (1) call the function *stop*() and (2) publish the generic event `INTERRUPTED`, i.e., the behavior has been forced to stop.

In summary, the possible events that can generate the execution of a behavior are: `WRONG_PROGRESS`, `GOAL_ACHIEVED`, `TIME_OUT`, and `INTERRUPTED`.

4.2. The behavior coordinator

An important issue for behavior execution is the coordination of the activation of multiple and interacting behaviors that can operate simultaneously. In our design, this coordination is performed by a centralized process, called behavior coordinator, that knows in advance the potential conflicts and avoids the concurrent execution of incompatible behaviors. For example, certain behaviors that use the same actuators (e.g., flight maneuvers controlled by rotors) are incompatible, i.e., only one behavior can be performed at any given moment.

The behavior coordinator activates a behavior taking into account the consistency with the environment and with other active behaviors. When a behavior is requested to be active, the coordinator analyzes the following: (1) checks if the behavior has conflicts with the current situation (this is directly asked to the behavior process), and (2) checks group consistency, i.e., checks if the activation of the behavior is consistent with other behaviors that are already active (e.g., the behavior *keep hovering* and the behavior *go to point* cannot be active at the same time).

We assume that behaviors are requested to be active *by deliberation* (e.g., during the interpretation of a mission plan or directly by the operator). But behaviors can also be requested to be active *by reaction* to certain situations. For example, Interrap [Müller, 1996] uses the concept of *reactors* for this type of behaviors. An example of this is the behavior *land* that is activated when the battery charge is low. When the behavior *take off* finishes (or when the behavior *go to point* finishes) the behavior *keep hovering* is automatically activated to be sure that the flight is correctly controlled.

These two activation request methods (deliberative or reactive) are a potential source of conflict when incompatible behaviors are requested to be active at the same time. To solve these conflicts we use a priority value for the reactive activation request:

- *Lower*. The reactive activation request of the behavior has lower priority than the deliberative activation request. This corresponds to a category of behaviors that must be active unless the operator activates other incompatible behaviors.
- *Higher*. The reactive activation request of the behavior has higher priority than the deliberative activation request. This category corresponds to behaviors for emergency situations that must be active, no matter what the operator says.

Thus, the order of priority is: *reaction with higher priority* > *deliberation* > *reaction with lower priority*. In the presence of two requests of the same order of priority, we apply the control strategy of *recency*, i.e., the most recent request has higher priority.

Group consistency is analyzed by a different process (the behavior specialist). If the current situation is not compatible with the requested behavior, the request is rejected (see algorithm 4). If an incompatible behavior with high priority is active, the coordinator rejects the request. If an incompatible behavior with low priority is active, the coordinator stops the active behavior.

Algorithm 4: Activate behavior *b*

```
1.  check if the activation of behavior b is compatible with current situation
2.  if (behavior b is not compatible with current situation)
3.  then
4.     reject the request
5.  else
6.     if (behavior b is not compatible with an active behavior of higher priority)
7.     then
8.        reject the request
9.     else
10.       if (behavior b is not compatible with a set of active behaviors A of lower priority or equal priority)
11.          then stop all behaviors in A
12.       start behavior b
```

The behavior coordinator activates sequentially behaviors as they are requested. In addition, the coordinator reacts to events generated by the execution of an activated behavior (GOAL_ACHIEVED, TIME_OUT, WRONG_PROGRESS or INTERRUPTED). These events are received through the ROS topic behavior event. When the coordinator receives this event, this means that the behavior has finished the execution (with success or failure). Thus, the coordinator removes the behavior from the list of active behaviors and activates all default behaviors that are compatible with the active behaviors (algorithm 5).

Algorithm 5: Behavior event handling (behavior *b*)

```
1.  A ← set of active behaviors
2.  A ← A - {b} /* remove behavior b from the list of active behaviors A */
3.  C ← set of all behaviors that could be requested to be active by reaction
4.  C ← C - A  /* remove behaviors in C that are already active */
5.  for each behavior b_i in C: /* loop to filter incompatible behaviors in C */
6.    if (b_i has lower priority than deliberative activation request)
7.    then
8.      for each behavior b_j in A (stop this loop if one incompatibility is detected)
9.        if (b_i is not compatible with b_j )
10.       then  C ← C - {b_i}
11. for each behavior b_k in C: /* loop to active only behaviors of C that satisfy their conditions  */
12.   if (default activation condition of b_k is satisfied) /* this takes time because consults belief manager, so it must be optimized */
13.   then
14.     activate behavior b_k
15.     if (b_k is successfully activated) then A ← A ∪ {b_k}
16. update active behaviors with the value of A
```

Note that a limitation of our design is that it does not accept the simultaneous execution of two occurrences of the same process. For example, if the robot has two arms, it is not possible to have one behavior called *move arm* for moving both arms simultaneously. Two different behaviors with different names would be needed: *move left arm* and *move right arm*. Another solution is to have a specific sub-coordinator as a single process, specialized in a category of behaviors, that coordinates the concurrent execution of a set of processes (e.g., *moving arms coordinator*).

4.3. The behavior specialist

The behavior specialist is a process that centralizes information about how to use each behavior and information about the expected evolution of each behavior. For this purpose, the behavior specialist use a file, called *behavior catalog* written in YAML format with information about each behavior such as the correct names, arguments, allowed values and compatibility.

With this information, the behavior specialist can check whether the activation request for a behavior is correct or not. For example, the numerical value for a particular argument may be out of the limits of the allowed values. The behavior specialist can also check whether particular groups of behaviors are compatible to be active simultaneously or not.

```
behavior_descriptors:

  - behavior: GO_TO_POINT
    timeout: 120
    category: goal_based
    incompatible_lists: [motion_behaviors]
    capabilities: [SETPOINT_BASED_FLIGHT_CONTROL, PATH_PLANNING]
    arguments:
      - argument: COORDINATES
        allowed_values: [-100,100]
        dimensions: 3
      - argument: RELATIVE_COORDINATES
        allowed_values: [-100,100]
        dimensions: 3

  - behavior: ROTATE
    incompatible_lists: [motion_behaviors]
    capabilities: [SETPOINT_BASED_FLIGHT_CONTROL]
    arguments:
      - argument: ANGLE
        allowed_values: [-360,360]

  - behavior: KEEP_MOVING
    category: recurrent
    incompatible_lists: [motion_behaviors]
    capabilities: [SETPOINT_BASED_FLIGHT_CONTROL]
    arguments:
      - argument: SPEED
        allowed_values: [0,30]
      - argument: DIRECTION
        allowed_values: [BACKWARD, FORWARD, UP, DOWN, LEFT, RIGHT]
...

behavior_lists:

  - list: motion_behaviors
    behaviors:
      - TAKE_OFF
      - LAND
      - FLIP
      - KEEP_HOVERING
      - FOLLOW_OBJECT_IMAGE
      - START_MOVING
      - START_HOVERING
      - GO_TO_POINT
      - ROTATE
...

reactive_activation:

  - behavior: KEEP_HOVERING
    condition: flight_state(self, FLYING)
    priority: lower

  - behavior: LAND
    condition: charge(battery, ?X), less_than(?X, 10)
    priority: higher

  - behavior: SELF_LOCALIZE_BY_ODOMETRY
    priority: lower
...
```

Figure 6: Behavior description in the behavior catalog

In the catalog, we can specify when to activate automatically a behavior (reactive activation) including the two options for priority (lower or higher). It is possible to include conditions expressed with belief expressions.

The catalog also includes the set of processes that are required to execute a behavior. To express this in a modular way, we use *capabilities*. A capability represents a particular robot's feature that is achieved by executing certain processes. A process may belong only to one capability. Table 4 shows examples of potential capabilities that can be used in aerial robotics. We may have different types of capabilities such as perception capabilities (the robot is able to perceive certain characteristics like colors, voice, visual markers, etc.), situation awareness capabilities (the robot is able to generate certain situation awareness abstractions like self localization, obstacles detection, or map building), and flight maneuver capabilities (the robot has certain capabilities that help in flight maneuvers like plan trajectories or limit extreme movements).

Capability	**Description**
VISUAL_MARKERS_RECOGNITION	The robot recognizes visual markers
SELF_LOCALIZATION_BY_ODOMETRY	The robot localizes its own position by using odometry
SELF_LOCALIZATION_BY_VISUAL_MARKERS	The robot localizes its own position by using visual markers
SELF_LOCALIZATION_BY_LIDAR	The robot localizes its own position by using LIDAR
SELF_LOCALIZATION_BY_GPS	The robot localizes its own position by using GPS
COLOR_RECOGNITION	The robot recognizes colors
SHAPE_RECOGNITION	The robot recognizes shapes
SPEECH_RECOGNITION	The robot recognizes voice commands from the operator
HELIPAD_RECOGNITION	The robot is able to recognize helipads
VISUAL_SERVOING	The robot controls the flight by vision
SETPOINT_BASED_FLIGHT_CONTROL	The robot controls the flight using reference values (setpoints)
SPEECH_GENERATION	The robot generates spoken words
CAMERA_CONTROL_FOR_PICTURES	The robot controls the camera to take pictures
CAMERA_CONTROL_FOR_VIDEOS	The robot controls the camera to record videos
LIGHT_CONTROL	The robot controls the lights
ARM_CONTROL	The robot controls the arm
PATH_PLANNING	The robot generate paths that avoid obstacles
2D_MAP_GENERATION	The robot builds a 2D map of the environment
3D_MAP_GENERATION	The robot builds a 3D map of the environment
USER_INTERACTION_TO_REQUEST_IMAGE	The robot can request to the operator to select a part of the camera image
USER_INTERACTION_TO_DISPLAY_IMAGE	The robot displays on the ground station monitor the camera image
SOCIAL_COMMUNICATION	The robot communicates with other drones
OPERATION_WITH_AR_DRONE	The robot is supported by the physical platform AR Drone 2.0
OPERATION_WITH_RVIZ_SIMULATOR	The robot is supported by a simulator based on RViz
OPERATION_WITH_GAZEBO_SIMULATOR	The robot is supported by the Gazebo simulator

Table 4. Examples of capabilities.

Figure 7 shows how capabilities are described in the behavior catalog. The information of each capability is written in a descriptor that includes the name and the set of processes that must be running to support the capability. In addition, it is possible to write ROS service calls to adjust the execution of certain processes. On the other hand, each behavior descriptor includes the required capabilities for the behavior.

```
capability_descriptors:

  - capability: SETPOINT_BASED_FLIGHT_CONTROL
    process_sequence: [droneTrajectoryController]
    incompatible_capabilities: [VISUAL_SERVOING]

  - capability: PATH_PLANNING
    process_sequence: [droneTrajectoryPlanner, droneYawPlanner]

  - capability: VISUAL_SERVOING
    process_sequence: [trackerEye, open_tld_translator, droneIBVSController]
    incompatible_capabilities: [SETPOINT_BASED_FLIGHT_CONTROL]

  - capability: DYNAMIC_SELF_LOCALIZATION_MODE
    process_sequence: [self_localization_mode_selector]

  - capability: SELF_LOCALIZATION_BY_ODOMETRY
    permanent_active: yes
    process_sequence: [droneOdometryStateEstimator]

  - capability: SELF_LOCALIZATION_BY_VISUAL_MARKERS
    process_sequence:
      - droneLocalizer

  - capability: OBSTACLE_DETECTION_BY_VISUAL_MARKERS
    process_sequence:
      - droneObstacleDistanceCalculator
      - droneObstacleProcesser

  - capability: VISUAL_MARKERS_RECOGNITION
    process_sequence:
      - droneArucoEyeROSModule
```

Figure 7: Description of capabilities

The behavior specialist may also help to verify if a mission plan is feasible, analyzing if a given sequence behaviors can be executed. This analysis does not perform a complete and detailed verification, since many details are only known during the actual execution. The main purpose of this verification is to detect in advance evident incorrect sequences of behaviors based on physical common sense. For example, this process can help to detect that it is not correct to request to land and, then, to request again to land. For this purpose, the behavior catalog can include the following representation:

- There is a set of state machines, where each state machine is associated to a particular physical phenomenon. Examples of physical phenomena are: light (the light of the robot that can be on or off) and flight (flight state of the robot).
- Each state machine includes states and transitions. Each state represents a qualitative situation of physical phenomena. For example, the states for the physical phenomenon light can be: on or off. The states for physical phenomenon flight can be: hovering, landed, moving. Each transition corresponds to a valid activation of a behavior. If the transition is not present, the behavior activation is not possible.

With this representation, the behavior specialist can check if a sequence of transitions is valid using the state machines. The procedure receives a sequence of behaviors and applies the transitions using the set of the state machines to verify that the activations are valid.

It is very important to keep the consistency between the behavior catalog (where capabilities are related to behaviors) and the software architecture configured for the aerial robot. When the software architecture is configured by the developer, the set of processes are declared to be launched and ready to be executed (using ROS launchers). It is important that all the potential processes defined in the catalog must be declared in the launchers. To ensure that the architecture

and catalog are consistent, the behavior specialist verifies that all the processes of the behavior catalog are consistent with the processes declared in the launchers. This verification is done while the catalog is loaded. If an inconsistency is detected, the behavior specialist shows a warning message to the operator informing that the capability is not supported by the architecture and continue the execution.

```
behavior_transitions:
  - behavior: TAKE_OFF
    physical_phenomenon: flight
    initial_state: landed
    final_state: hovering

  - behaviors: [GO_TO_POINT, START_HOVERING]
    physical_phenomenon: flight
    initial_state: [hovering, moving]
    final_state: hovering

  - behavior: START_MOVING
    physical_phenomenon: flight
    initial_state: [hovering, moving]
    final_state: moving

  - behavior: LAND
    physical_phenomenon: flight
    initial_state: [hovering, moving]
    final_state: landed

  - behavior: TURN_LIGHT
    argument: MODE
    argument_value: ON
    phenomenon: light
    initial_state: off
    final_state: on
```

Figure 8: Example representation of state machines to verify the feasibility of a sequence of behaviors.

4.4. The resource manager

The role of the resource manager is to run the processes that support the execution of behaviors, trying to use efficiently the limited resources of the robot. As mentioned before, the processes of a behavior are grouped in capabilities (each process may belong only to one capability). When a behavior is activated, its corresponding capabilities are activated automatically by starting the corresponding processes. We assume that human operators do not specify directly when to activate or deactivate capabilities (e.g., in a mission plan). Otherwise, there can be conflicts between what the operator says and what a behavior needs.

The activation of capabilities is associated to the increase of resource consumption (memory space, processing time, battery charge) so it is important to deactivate automatically unnecessary capabilities when it is possible. Since a capability may be used by different concurrent behaviors, the capability can be deactivated only when it is not used by any behavior. For this purpose, the resource manager keeps updated a number of references for each capability. When a capability has zero references, the resource manager can stop its processes. However the actual stop of processes is not done immediately to keep the processes running, avoiding unnecessary stops when two consecutive behaviors use the same processes.

The resource manager has been designed to activate the processes taking into account also the existing faults and making decisions about alternative processes to start, for example, when a

resource is broken or already used (e.g., a right handed person can open a door with the left hand when the right hand is holding something). For example, if an acoustic altitude sensor fails, the manager can decide to use a pressure sensor to estimate altitude. Or, in a robot that has stereo vision, sonars and infrared sensors, the resource manager can decide what sensor to use according to the range detection (e.g., IR can detect at a sufficient range, stereo vision may be fast enough to work when the robot is moving, and sonars may produce reliable readings) [Murphy, 2000].

5. The belief management system

We consider a *belief* as a proposition about the world that the robot thinks is true (the world here refers to both the external world and the internal state of the robot). The robot uses beliefs as true facts while it reasons trying to adapt the execution of the mission plan to the current circumstances.

The belief management system stores the set of beliefs and keeps their consistency (with the world and with the other beliefs). Figure 9 shows the set of processes of this system. It includes a process for belief management and a set of processes that we call belief updaters.

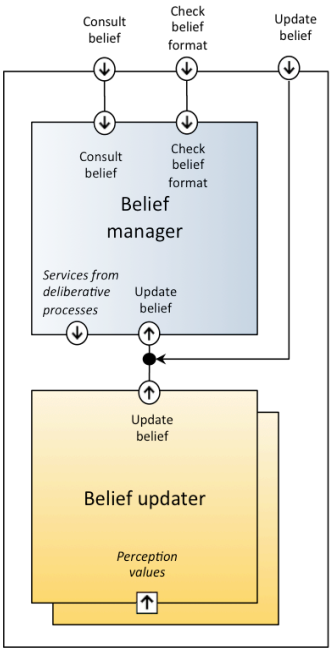

Figure 9: Architecture of the belief management system.

5.1. The belief manager

The goal of the belief manager is to manage the memory of beliefs accepting requests to add and delete beliefs and questions about the existence of beliefs (i.e., consult a belief expression). The belief manager ensures the consistency in the stored beliefs.

It is important to use a uniform and symbolic representation for beliefs that can be used (1) by the operator to write mission plans, and (2) by the robot to explain to the operator its own behavior. Beliefs should be based on notions used in the operator's language, to generate understandable descriptions. With symbolic beliefs, the robot may communicate more naturally to the operator its

own knowledge simulating a kind of conscious understanding of the current situation and its own performance.

It is important to note that only a small part of all the information used by the robot needs to be represented as beliefs. For example, the robot uses an environment map that may have detailed information about dimensions, limits, walls, obstacles, etc. However, it is not necessary to use beliefs to represent all the details of the map. The general rule is that certain information is represented as belief if the operator can use it to formulate a mission plan or to understand an unexpected behavior. For example, information is represented as belief if the mission plan uses this information to make decisions, or if the behavior uses this information as precondition to check its correct activation. During the execution, the size of the content of the belief memory is normally small (e.g., usually a few dozens of beliefs).

We represent beliefs using a logic-based approach with predicates. Table 5 shows examples of such predicates with the general format of *predicate*(*object*, *value*) or simpler forms such as *property*(*object*). The representation follows also an object-oriented approach. Objects are instances of a class. They can have attributes with values. We assume that the values of attributes defined for an object using triplets are *mutually exclusive*. For example, the belief `charge(battery, empty)` is incompatible with the belief `charge(battery, full)` because the values `empty` and `full` are mutually exclusive. If a particular relation does not have values that are not mutually exclusive, this must be explicitly stated as an exception. The belief manager keeps consistency among beliefs. When a belief is added, e.g., `charge(battery, empty)`, the incompatible beliefs are automatically retracted, e.g., `charge(battery, full)`.

Predicate	Description
`object(x, y)`	The object x is an instance of the class y.
`position(x, y)`	The object x is at the position y.
`name(x, y)`	The name of object x is y.
`flight_state(x, y)`	The aerial robot x is flight state y (e.g., landed or flying)
`code(x, y)`	The numerical code of x is y.
`color(x, y)`	The color of x is y.
`frequency(communication,x)`	The frequency of the communications is x.
`charge(battery, x)`	The charge of battery is x.
`carry(self, x)`	The own aerial robot carries the object x.
`image(x, y)`	The image of object x is y.
`temperature(air, x)`	The temperature of the air is x.
`visible(x)`	The object x is visible.
`stability(ground, x)`	The stability of the ground is x.

Table 5. Examples of predicates to represent beliefs about situation awareness.

As it was explained above, beliefs can be used to detect certain types of errors. For example, the assumptions of a behavior for its activation (e.g., the behavior *take off* can be activated only if the robot is landed) can be expressed as a set of preconditions represented with belief expressions.

Table 6 shows examples of such expressions. The expressions are verified to identify conflicts. When a conflict is detected, the robot can inform the operator what are the beliefs that do not verify the expression. We assume that the behavior does not have to check for conditions that are common for the majority of behaviors (e.g., `communication(wireless, ok)`, `charge_level(battery, ok)`, `visibility(camera, ok)`, `vibrations(self, none)`). This is done more efficiently, in a separate reactive process that verifies these events and reacts accordingly.

Behavior	Precondition
GO_TO_POINT	flight_state(self, ?x1), belong(?x1, [moving, hovering]) position(self,?x2), approximate_travel_time(?x2, ?y,?t), ?t < 30 minimum_distance_to_obstacle(?y, ?d), ?d > 3 *NOTE: ?y is the destination point*
FLIP	flight_state(self, ?x), belong(?x, [moving, hovering]) position(self, ?x), minimum_distance_to_obstacle(?x, ?y), ?x > 3 charge_level(battery, ?x), ?x >30
LAND	flight_state(self, ?x), belong(?x, [moving, hovering]) stability(ground, stable)

Table 6: Examples of belief expressions as preconditions for behavior execution.

Predicate	Description
belong(x, y)	The element x belongs to the list y
x > y, x < y, x = y, x >= y, x =< y	Comparison operators for numbers x and y.
approximate_travel_time(x, y)	The approximate travel time to go from the current position to x is y.
distance_to_obstacle(x, y)	The minimum distance to an obstacle from x is y.

Table 7. Examples of beliefs to represent the result of simple functions.

5.2. The belief updater process

Beliefs are updated periodically using information from sensors. This is done by processes called *belief updaters*. Each belief updater is a process specialized in updating a category of beliefs. Beliefs do not need to be updated at a high frequency, compared to the frequency used by controllers. It is enough a lower frequency to work with the executive system. Beliefs can be updated periodically at a certain frequency or they can be updated when they are consulted.

To indicate that an object is currently observed, we use the predicate `visible().` When an object is not observed, this predicate is retracted, but the rest of predicates related to this object are kept in memory (name, color, position, etc.). In general, beliefs should be stored in memory with a limited duration. The robot may assume that, after a period of time, something that was observed in the world is not longer true (based on common sense about the objects in the world). We call this the *temporal persistence* of a belief. The beliefs of each type of object have a certain persistence value according to its nature. For example, beliefs about the position of a moving object (e.g., a person, a car) should have a very low persistence and beliefs about the position of a static object (e.g., a wall, a door) should have a high persistence. A mobile object (e.g., a chair, a bucket) can have an intermediate value of persistence.

Based on the duration of missions that perform aerial robots (usually less than one hour), we can simplify the management of persistence with the practical assumption that beliefs may have one of two extreme values: permanent persistence or null persistence. By default, all objects have permanent persistence (the operator can express exceptions to this). Thus, the majority of beliefs are stored forever, unless incompatible beliefs are added. As an alternative to this approach, it is possible to use other more complex representations such as a probabilistic approach to model uncertainty about persistence, variable duration of persistence calibrated for each type of object, methods to forget non relevant beliefs, etc.

Beliefs can be anchored to percepts (e.g., to images). This is useful to learn to recognize objects. For example, the robot can memorize the image of a person together with a belief with the name. This can be used to recognize again the person and call her/him by her/his name. A simple method to learn this is by storing a belief that links the name and the image. As an alternative, instead of the image we could store the recognition process of the image (e.g., a trained neural network).

A problem related to this is to know if the perception is unique for the object. For example, if the robot has recognized a chair in the past and now the robot sees a similar image, is it the same chair? And also, is it important for the robot to know that it is the same chair? For our missions, we can assume that images have univocal interpretation. Therefore, if the robot recognizes the face of a person, the name is also recognized. For more complex missions, we can consider that certain objects don't have this property.

An active management of beliefs could modify the behavior of the robot to focus attention on certain objects. For example, in the presence of uncertainty between symbolic name and image, the robot can move the camera to have a better image in order to reduce uncertainty. Another example is related to the persistence of the position of an object, using an estimated duration of persistence. When the persistence time is close to the end, the robot can move the camera to see if the object continues in the same position.

6. Interpreters for mission plans

This section presents some examples of mission plan interpreters that illustrate how the execution engine can be used using different representation languages. We present here three interpreters used in Aerostack:

- *TML mission interpreter*. Executes a mission specified in the language TML (Task-based mission specification language).
- *Python-based mission interpreter*. Executes a mission specified in the language Python.
- *Behavior tree mission interpreter*. Executes a mission specified using behavior trees.

The TML mission interpreter uses a language with XML syntax called TML (Task-based mission specification language) that uses a hierarchical approach to describe how to decompose complex tasks of a mission plan into a sequence of atomic executable actions. TML belongs to the category of execution languages (e.g., the Plexil language [Verma et al., 2005]). TML is described in detail in some publications [Molina et al., 2016]. The following two sections describe the other two interpreters.

6.1. The interpreter of mission plans written in Python language

A mission plan can be represented in Python language using the behaviors and beliefs presented in this paper. This language is presented to the developer as an API (Application Programming Interface) with a library of functions (see next table).

The interpreter of this language can verify in advance (before the plan is executed) the correct format of the Python program. For example, the interpreter can verify that all the names of behaviors exist and all the beliefs expressions are correct.

With this approach, the operator can write the mission tasks directly in Python calling specific functions of the API library specialized in robotic actions. This is a very flexible approach but presents two main limitations: (1) it is difficult to verify the complete feasibility of the specification in advance, and (2) it is difficult to be used by users who are not familiar with computer languages.

Function	Description	Example of use
`executeBehavior(x, y)`	Executes a goal-based behavior *x* with arguments *y*, and waits until the behavior reaches the goal. This function returns the result of the execution. The result is one of the following values: {ACHIEVED_GOAL, TIME_OUT, WRONG_PROGRESS, FAILED_PRECONDITIONS}	`executeBehavior('GO_TO_POINT', point='(1, 2, 3)')` `result = executeBehavior('LAND')`
`activateBehavior(x, y)`	Activates a behavior *x* with arguments *y*. This function returns the result of the activation. The result is one of the following values: {OK, FAILED_PRECONDITIONS}	`activateBehavior('KEEP_HOVERING')`
`askBehavior(x, y)`	Executes a deliberative behavior *x* with arguments *y*, and returns a string with the result.	`path = askBehavior('GENERATE_PATH')`
`inhibitBehavior(x)`	Inhibits the activation of behavior *x*. We assume that this function always succeeds.	`inhibitBehavior('RECOGNIZE_ARUCO_MARKERS')`
`isActiveBehavior(x)`	Answers whether a behavior *x* is active (true) or not (false).	`isActiveBehavior('RECOGNIZE_ARUCO_MARKERS')`

`consultBelief(x)`	Returns a tuple (R1, R2), where R1 is a boolean indicating whether the belief expression *x* matches an instance in the belief memory or not, and R2 is a list (as a Python dictionary) with the value of the variables of belief expression *x* that unify such expression with an instance in the belief memory. Only the first instance that matches the expression is considered. If the belief expression does not match any instance in the belief memory R1 is false.	`success, unification =`
		`consultBelief('charge(battery, ?X)')`
		`if success:`
		`  x = unification['?X']`

`trueBelief(x)`	Returns true if the belief expression *x* is true or false in other case.	`trueBelief('object(?X, marker), code(?X, 3')`

Table 8: Functions of the Execution Engine API.

```
import executive_engine_api as api

def runMission():
  api.executeBehavior('TAKE_OFF')

  api.activateBehavior('PAY_ATTENTION_TO_VISUAL_MARKERS')

  success, unification = api.consultBelief('position(self, (?x,?y,?z))')

  if success:
     x, y, z = unification['x'], unification['y'], unification['z'],
  else:
     print "Position unknown"

  api.executeBehavior('GO_TO_POINT',point=[2,5,1.3])

  api.executeBehavior('GO_TO_POINT',point=[x,y,z])

  api.inhibitBehavior('PAY_ATTENTION_TO_VISUAL_MARKERS')

  api.executeBehavior('LAND')
```

Figure 10: Example program in Python for a mission plan.

6.2. The interpreter of mission plans represented with behavior trees

A behavior tree is a popular representation that was proposed in the computer gaming industry. In robotics, behavior trees have been used recently [Marzinotto et al 2014] [Colledanchise, Ogren, 2014] and specifically for UAVs [Ögren 2012] [Klöckner, 2013]. The goal of the MBT mission interpreter is to execute a mission plan described as a behavior tree (MBT - Mission plan specification with behavior trees). Figure 11 shows an example of behavior tree for a particular mission (mission 7 of IARC).

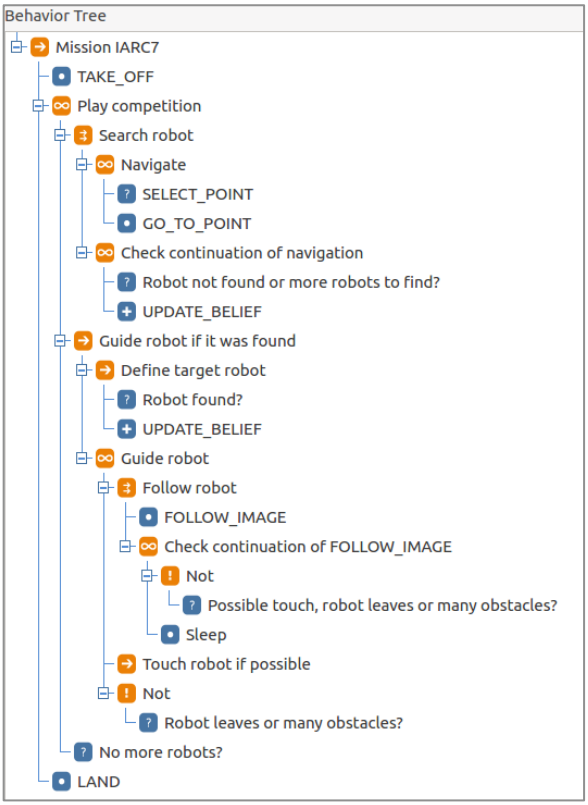

Figure 11: Example of behavior tree.

In short, the interpreter interacts with the execution engine in the following way. Intermediate nodes of the tree establish the control regime (e.g., a sequence, a loop, etc.). The leaf nodes of the tree may correspond to: behaviors or conditions represented with belief expressions. The interpreter of the behavior tree travels through the nodes following the control regime established by intermediate nodes and, when a leaf node is reached, the interpreter interacts with the execution engine in the following way: (1) *behavior*, if the leaf node is a behavior, the interpreter requests the execution engine to execute such a behavior, (2) *condition*, if the leaf node is a condition, the interpreter consults belief expression to the execution engine.

The behavior representation makes it easier for operators that are not familiar with computer languages to formulate a mission plan. In contrast, this solution is less flexible than the solution based on a programming language like Python. The implementation of several approaches in Aerostack for mission planning is easier thanks to the execution engine. Having multiple methods to formulate mission plans is appropriate to offer alternative solutions to be chosen by operators according to their preferences.

7. Conclusions

In this paper, we have presented the technical details of an execution system that has been designed to help simplify the specification of mission plans for aerial robot systems. This work has been done as part of our efforts to promote a more extensive use of the software framework Aerostack.

The proposal follows a behavior-based approach that simplifies the description of mission plan with a uniform representation. The approach is supported by a general architecture conceived as an execution engine with two main systems: one system to facilitate the execution of behaviors and another one to manage a memory of beliefs that are represented with predicates.

The system presented in this paper is based on solutions from the current state of the art in artificial intelligence and robotics. The main contribution of this work is an original design conceived to improve the usability of Aerostack. Thus, this paper can be useful to understand reasons that support the design of Aerostack. Compared to other execution systems, our design has been conceived using modern software technology (e.g., ROS, Linux, C++, etc.) to achieve acceptable levels of efficiency, as it is required in aerial robotics. The design of the execution engine is not committed with Aerostack, so it can be reused as a model for other different robot architectures.

The description of the execution engine presented in this paper corresponds to its specification and general design. In our research group, we are currently working on its implementation and its validation with real flight tests. As a result of this, this general design could be refined to cope with certain issues that might have not been considered in the initial version. Once the implementation is completed, we plan to make the programs freely available as part of a new release of the Aerostack open-source project (www.aerostack.org).

Acknowledgements

The development of Aerostack has been partially supported by the Spanish Ministry of Economy and Competitiveness through the project VA4UAV (Visual autonomy for UAV in Dynamic Environments) reference DPI2014-60139-R.

The work presented in this paper has been developed in the context of the Aerostack open-source project in the the research group CVAR (Computer Vision and Aerial Robotics) in the Technical University of Madrid (UPM). The following members of CVAR are the persons in charge of building the programs that implement and validate the design presented in this paper: Alberto Camporredondo, Guillermo de Fermin, Carlos Valencia, Jorge Pascual and Rafael Artiñano.

References

Allgeuer, P., Behnke, S. (2013): "Hierarchical and state-based architectures for robot behavior planning and control". In Proceedings of 8th Workshop on Humanoid Soccer Robots, IEEE-RAS Int. Conf. on Humanoid Robots, Atlanta, USA (pp. 3-5).

Brook, R. A. (1986): "A Robust Layer Control System for a Mobile Robot", IEEE Journal of Robotics and Automation RA-2, 14-23.

Brooks, R. A. (1990): "The Behaviour Language; User's Guide," MIT AI Lab.

Colledanchise M. and Ogren P. (2014): "How behavior trees modularize ro- bustness and safety in hybrid systems". In Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on, pages 1482–1488. IEEE, 2014.

Doherty, P., Heintz, F., Landén, D. (2010): "A distributed task specification language for mixed-initiative delegation". In International Conference on Principles and Practice of Multi-Agent Systems (pp. 42-57). Springer Berlin Heidelberg.

Firby, R. J. (1987): "An investigation into reactive planning in complex domains". In AAAI (Vol. 87, pp. 202-206).

Firby, R. J. (1989): "Adaptive execution in complex dynamic worlds". Doctoral dissertation, Yale University.

Firby, R. J., Slack, M. G. (1995): "Task execution: Interfacing to reactive skill networks". In AAAI Spring Symposium.

Gat, E. (1998): "On three-layer architectures". Artificial intelligence and mobile robots, 195, 210.

Gat, E., Dorais, G. (1994): "Robot navigation by conditional sequencing". In Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on (pp. 1293-1299). IEEE.

Gat, E. (1997). "ESL: A language for supporting robust plan execution in embedded autonomous agents". In Aerospace Conference, 1997. Proceedings., IEEE (Vol. 1, pp. 319-324). IEEE.

Klöckner, A., van der Linden, F., Zimmer, D. (2014). "The modelica behavior trees library: Mission planning in continuous-time for unmanned aircraft". In Proceedings of the 10th International Modelica Conference, number 96, pages 727–736.

Konolige, K. (1997). "Colbert: A language for reactive control in Sapphira," in KI-97: Advances in Artificial Intelligence, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 31–52.

Kortenkamp, D., Simmons, R., Brugali, D. (2016). "Robotic systems architectures and programming". In Springer Handbook of Robotics (pp. 283-306). Springer International Publishing.

Loetzsch, M., Risler, M., & Jungel, M. (2006): "XABSL-a pragmatic approach to behavior engineering". In Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on (pp. 5124-5129). IEEE. http:// www.xabsl.de

MacKenzie, D. (1997): "A design methodology for the configuration of behavior-based mobile robots". Ph.D. dissertation, Georgia Institute of Technology, GA, USA.

Marzinotto, M., Colledanchise, M., Smith, C., Ögren, P. (2014): "Towards a unified behavior trees framework for robot control". In Robotics and Automation (ICRA), 2014 IEEE International Conference on, pages 5420–5427. IEEE, 2014.

Mataric, M. (1994): "Interaction and intelligent behavior". Ph.D. thesis, MIT, EECS.

Michaud, F., Nicolescu, M. (2016). "Behavior-based systems". In Springer Handbook of Robotics. Springer International Publishing.

Molina, M., Díaz Moreno, A., Palacios, D., Suárez Fernández, R., Sánchez López, J. L., Sampedro Pérez, C., Bavle H., Campoy Cervera, P. (2016): "Specifying complex missions for aerial robotics in dynamic environments". The International Micro Air Vehicle Conference and Competition (IMAV 2016), Beijing, China.

Molina, M., Frau, P., Maraval, D., Sanchez-Lopez, J.L., Bable, H., Campoy, P. (2017): "Human-Robot Cooperation in Surface Inspection Aerial Missions". The International Micro Air Vehicle Conference and Competition, IMAV 2017, Toulouse, France.

Müller, J. P. (1996): "The design of intelligent agents: a layered approach". Springer Science & Business Media (Vol. 1177).

Murphy, R. (2000): "Introduction to AI robotics". MIT press.

Nicolescu, M. N., Matarić, M. J. (2002): "A hierarchical architecture for behavior-based robots". In Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1 (pp. 227-233). ACM.

Ögren, P. (2012): “Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees”. In AIAA Guidance, Navigation and Control Conference, Minneapolis, Minnesota, 13 - 16 August 2012. AIAA. AIAA 2012-4458.

Ridao, P., Yuh, J., Batlle, J., Sugihara, K. (2005): “On AUV Control Architecture”. In Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000), volume 2, pages 855–860.

Roberts, G. N., Sutton, R., Allen, R. (2993). “Guidance and control of underwater vehicles”. Elsevier Science and Technology, IFAC Proceedings Volumes(1):1–40.

Risler, M. (2009): “Behavior control for single and multiple autonomous agents based on hierarchical finite state machines,” PhD Dissertation. Fortschritt-Berichte VDI, Technische Universitt Darmstadt.

Molina, M., Bavle, H., Sampedro, C., Campoy, P. (2017): “A Multi-Layered Component-Based Approach for the Development of Aerial Robotic Systems: The Aerostack Framework”. Journal of Intelligent & Robotic Systems, 1-27.

Sanchez-Lopez, J.L., Suarez-Fernandez, R. A., Bavle, H., Sampedro, C., Molina, M., Pestana, J., Campoy, P. (2016): “AEROSTACK: An Architecture and Open-Source Software Framework for Aerial Robotics”. ICUAS 2016, Arlington, USA.

Sheridan, T. B. (1992): “Telerobotics, automation, and human supervisory control”. MIT Press.

Simmons, R., Apfelbaum, D. (1998): “A task description language for robot control”. In Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on (Vol. 3, pp. 1931-1937).

Suárez-Fernández, R. A., Sanchez-Lopez, J. L., Sampedro, C., Bavle, H., Molina, M., & Campoy, P. (2016). “Natural user interfaces for human-drone multi-modal interaction”. In Unmanned Aircraft Systems (ICUAS), 2016 International Conference on (pp. 1013-1022). IEEE.

Verma, V., Estlin, T., Jónsson, A., Pasareanu, C., Simmons, R., & Tso, K. (2005): “Plan execution interchange language (PLEXIL) for executable plans and command sequences”. In Proceedings of the 9th International Symposium on Artificial Intelligence, Robotics and Automation in Space.

Appendix A: Library of behaviors for aerial robotics

This section describes a library of behaviors for aerial robotics that we designed for our execution engine. The library of behaviors has been designed considering part of quality principles (Table 1). For example, the principle of orthogonality says that two behaviors are orthogonal if they do not interfere with one another, each inducing no side-effects in the other [Michaud, Nicolescu, 2016]. A language is said to be orthogonal if it allows the programmer to mix these constructs freely. In our library, this can be achieved only partially because there are certain behaviors that cannot execute simultaneously because they use the same effectors (e.g., keep hovering and go to point).

Principle	Description
Orthogonality	Behaviors are orthogonal if there are no restrictions on how they may be combined.
Parsimony	The number of behaviors should not be multiplied needlessly.
Necessity	Each behavior achieves a goal that cannot be achieved by other behavior [Michaud, Nicolescu, 2016].
Sufficiency	Each behavior is sufficient for achieving the goals mandated for the controller [Michaud, Nicolescu, 2016].
Generality	The library should be general to be applicable to as many different situations as possible.
Clarity	The name of the behavior should be as clear as possible, to avoid the need to explain its meaning.
Conciseness	The names used should express much in few words.
Simplicity	The name and arguments of behaviors should not be complex. The simplest of several expressions is to be preferred.
Stability	The library should be stable in time.
Scalability	The library should be able to grow, including with new behaviors, without losing its quality.

Table A.1: Quality principles for the design of a library of behaviors.

In our library of behaviors, we distinguish three basic categories:

- *Goal-based behaviors*. These behaviors are defined to reach a final state (attain a goal). Examples of these behaviors are flight maneuvers (related to rotors) such as simple flight maneuvers (take off, land) or complex flight maneuvers (go to a point, move in circles). This category also includes behaviors related to other effectors such as: sound (say sentence), light (turn on-off), camera (take photo or video), dropping mechanism (to drop an item), hand/arm (to grasp an item), moving camera (turn camera, look at a point), etc.

- *Recurrent behaviors*. These behaviors perform an activity recurrently or maintain a desired state. For example, some communication behaviors belong to this category (speak up, show video image to the operator). Other examples are: data storage behaviors (build a map, record a video), attention behaviors (pay attention to colors, pay attention to voice commands, etc.), etc.

- *Deliberative behaviors*. These behaviors correspond to deliberation tasks such as planning tasks. The result of these behaviors is either returned by the behavior event or stored in the memory of beliefs. A deliberative behavior can find a result successfully or it can fail due to, for example, the current situation (e.g., planning a path can be impossible if there is a barrier of obstacles).

Recurrent behaviors differ from goal-based behaviors in the following way. A goal-based behavior defines implicitly a goal to be achieved in terms of a final state. The final state can be formulated with a set of conditions about the state of the world. The behavior is deactivated in the moment when these conditions are satisfied (or when it fails). For example, the goal of the

behavior go-to-point is to be at a certain point. The final state is defined as the situation when the robot position is the same as the desired position. The behavior finishes when this condition is satisfied. These behaviors fail if the goal is not reached in a limited period of time (timeout). All goal-based behaviors have a default timeout value that can be changed by the operator in a particular mission plan.

In contrast, recurrent behaviors don't define a final state, so it is not possible to determine when the behavior has finished. Recurrent behaviors can express a desired state to maintain or a permanent activity that can be either active or inactive, so they can be working without any limit of time. For example, the recurrent actions build-map and record-video can be active all the time during a mission. Recurrent behaviors could be activated under certain conditions:

- *Distance*. The behavior is only activated when the distance between the position of the robot and a certain point (x, y, z) is less than certain value. Attribute: value (meters).

- *Delay*. The behavior is only activated after a number of seconds. Attribute: value (seconds).

- *Yaw*. The behavior is only activated for a particular yaw. Attribute: min value (degrees), max value (degrees).

Deliberative behaviors are used by the execution engine to create a uniform and robust interface with deliberative processes using correct symbolic names and correct arguments and values. For example, the behavior `GENERATE_PATH_FREE_OF_OBSTACLES` uses the deliberative process called *path planner* that finds a path free of obstacles to go from a point *x* to a point *y*. Another example is the behavior `GENERATE_NEXT_POINT_TO_EXPLORE` that may use the process *exploration planner*. This process finds the next point to explore according a certain strategy. Some of these processes may have memory (e.g., remember the explored area to generate a new point to explore) and, therefore, they can operate sequentially.

This type of beliefs can help to formulate general descriptions about how to do complex behaviors using simpler behaviors. For example, the high level action *go to point y* could be formulated as a complex behavior in the following way:

1. Assuming that the robot is in position x, consult the belief `path_free_of_obstacles(x, y, z).` This can execute the process *path planner* to generate the value of the variable z with the path.
2. Execute the behavior *follow path z.*
3. If there is an obstacle during the execution, go to 1.

Another example is the complex behavior *explore spatial region x* that could be formulated in the following way:

1. Consult the belief `next_point_to_explore(x, y).` This can execute the process *exploration planner* to generate the value of the variable *y* with the point to explore.
2. Execute the behavior *go to point y.*
3. If the region is not completely explored, go to 1.

The following tables show the design of our library of behaviors. Tables present sets of behaviors that belong to a certain categories such as general flight maneuvers, maneuvers guided by visual references, communication, planning behaviors, etc.

Behavior	Type	Description	Arguments
TAKE_OFF	Goal-based	The robot takes off vertically from the surface to the normal altitude. If the altitude argument is not given, the robot reaches a default altitude.	ALTITUDE (meters)
LAND	Goal-based	The robots lands vertically in the current position.	
KEEP_HOVERING	Recurrent	The robot keeps hovering. Hovering is a maneuver in which the robot is maintained in nearly motionless flight over a reference point at a constant altitude and on a constant heading. This behavior does not avoid moving obstacles.	
KEEP_MOVING	Recurrent	The robot keeps moving at a constant speed in some direction (forward, backward, upward, downward}. If the speed value is not given a default value is considered. This behavior does not avoid obstacles.	DIRECTION {FORWARD, BACKWARD, UPWARD, DOWNWARD} SPEED (m/sec)
ROTATE	Goal-based	The robot rotates some degrees in a certain axis (yaw, pitch, roll}	AXIS {YAW, ROLL, PITCH} ANGLE (degrees), RELATIVE_ANGLE (degree), DIRECTION {CLOCKWISE, COUNTERCLOCKWISE}
GO_TO_POINT	Goal-based	The robot goes to a point avoiding obstacles. Time is expressed with the number of seconds since the mission starts. Spline means that the robot flies smooth paths both vertically and horizontally instead of straight lines.	COORDINATES (x,y,z in meters), RELATIVE_COORDINATES (x,y,z) POINT_TO_LOOK_AT_THE_END (x,y,z), YAW_ANGLE_AT_THE_END (degrees), TIME (seconds), POINT_TO_LOOK_DURING_FLIGHT (x,y,z), SPLINE (yes, no),
FLIP	Goal-based	The robot performs a flip movement.	DIRECTION {RIGHT, LEFT, FRONT, BACK}
FOLLOW_OBJECT_IMAGE		The robot follows a moving object image, keeping a certain constant distance between the drone and the object.	IMAGE (image)
FOLLOW_PATH	Goal-based	The robot tries to follow a path defined as a sequence of points.	PATH (sequence of points (x,y.z))
LOOK_AT_A_POINT	Goal-based	The robot looks at a certain point, i.e. the robot rotates (yaw and pitch) to see the point through the front camera.	COORDINATES (x, y, z in meters),
KEEP_LOITERING	Recurrent	The robot keeps moving in circles or squares . Avoid obstacles is active by default.	SHAPE (CIRCLE, SQUARE) DELAY (seconds), CLOCKWISE (true/false), RADIUS (meters, it can be zero), SIDE (side length of the square in meters, different from zero), AVOID_OBSTACLES (yes, no).
TRACK_OBJECT_IMAGE	Recurrent	The robot keeps looking to an object image. The robot keeps its position but rotates to keep looking at a moving image.	IMAGE (image)
ALIGN_OVER_IMAGE	Goal-based	The robot aligns vertically over an image.	IMAGE (image)
LAND_ON_MOVING_ PLATFORM	Goal-based	The robot lands on a moving platform.	
TOUCH_MOVING_OBJECT	Goal-based	The robot touches a moving object.	
TAKE_OFF_FROM_MOVING_ PLATFORM	Goal-based	The robot takes off from a moving platform.	

Table A.2: Motion behaviors based on different flight maneuvers.

Category	Behavior	Type	Description	Arguments
Understanding	SELF_LOCALIZE_BY_VISUAL_MARKERS	Recurrent	The robot self localizes using visual markers.	
Understanding	PAY_ATTENTION_TO_VISUAL_MARKERS	Recurrent	The robot pays attention to visual markers. The recognized colors are stored as beliefs.	MARKER {ARUCO, …}
Understanding	PAY_ATTENTION_TO_COLORS	Recurrent	The robot pays attention to the images to recognize colors. The recognized colors are stored as beliefs.	
Understanding	PAY_ATTENTION_TO_SHAPES	Recurrent	The robot pays attention to the images to recognize shapes. The recognized shapes are stored as beliefs.	
Understanding	VERIFY	Deliberative	The robot checks if a condition is satisfied. The result is true if the condition is satisfied. Otherwise, the result is false.	MARKER_IS_OBSERVED (integer) HOVERING(true, false) CURRENT_SPEED(integer) OBSTACLE_DISTANCE(integer) VOICE_COMMAND(text)
Planning	GENERATE_PATH_FREE_OF_OBSTACLES	Deliberative	The robot generates a path free of obstacles to go from the current position to a certain destination. The result is a list of 3D points such as: [[2.5, 3.2, 1.5], [1.2, 3.3, 1.5]]. If the robot is not able to generate a path, it generates the empty value.	DESTINATION (x,y,z in meters),
Planning	GENERATE_NEXT_POINT_TO_EXPLORE	Deliberative	The robot generates the next point to explore a spatial region. The result is a 3D point such as: [3.5, 2.3, 1.5]. If the there are not more points to explore, it generates the empty value.	REGION (list of x,y,z in meters),
Communication	BROADCAST_MESSAGE	Goal-based	The robot broadcasts a message to be received by other drones.	MESSAGE (string)
Communication	NOTIFY_OPERATOR	Goal-based	The robot sends a message to be received by the operator	MESSAGE (string)
Communication	ASK_OPERATOR	Goal-based	The robot ask a question to the operator	QUESTION (string), POSSIBLE_ANSWERS(list of strings)
Communication	ASK_FOR_OBJECT_IMAGE	Goal-based	The robot ask the operator for an object image.	
Communication	DISPLAY_CAMERA_IMAGE	Recurrent	The robot shows to the operator the camera image. This requires, for example, to activate the capability front_camera_sensing (if it is optional).	
Communication	SPEAK_UP	Recurrent	The robot says out loud the content (tasks, etc.)	CONTENT {TASKS, ...}
Communication	SAY	Goal-based	The robot says a sentence out loud.	SENTENCE(string)
Data recording	BUILD_MAP	Recurrent	The robot builds a map.	DIMENSION {TWO, THREE}
Data recording	RECORD_VIDEO	Recurrent	The robot records a video.	
Data recording	TAKE_PHOTO	Goal-based	The robot takes a photo.	
Configuration	TURN_LIGHTS	Goal-based	The robot changes the status of the light	VALUE {ON, OFF}
Manipulation	DROP_ITEM	Goal-based	The robot drops an item.	

Table A.3: Other behaviors.

Appendix B: Example of behavior catalog

This appendix shows a complete example of behavior catalog in YAML language.

```
default_behavior_values:
  timeout: 15
  category: goal_based

behavior_descriptors:
  - behavior: TAKE_OFF
    incompatible_lists: [motion_behaviors]

  - behavior: LAND
    incompatible_lists: [motion_behaviors]

  - behavior: GO_TO_POINT
    timeout: 120
    incompatible_lists: [motion_behaviors]
    capabilities: [SETPOINT_BASED_FLIGHT_CONTROL, PATH_PLANNING]
    arguments:
      - argument: COORDINATES
        allowed_values: [-100,100]
        dimensions: 3
      - argument: RELATIVE_COORDINATES
        allowed_values: [-100,100]
        dimensions: 3

  - behavior: ROTATE
    incompatible_lists: [motion_behaviors]
    capabilities: [SETPOINT_BASED_FLIGHT_CONTROL]
    arguments:
      - argument: ANGLE
        allowed_values: [-360,360]

  - behavior: KEEP_MOVING
    category: recurrent
    incompatible_lists: [motion_behaviors]
    capabilities: [SETPOINT_BASED_FLIGHT_CONTROL]
    arguments:
      - argument: SPEED
        allowed_values: [0,30]
      - argument: DIRECTION
        allowed_values: [BACKWARD, FORWARD, LEFT, RIGHT]

  - behavior: FOLLOW_OBJECT_IMAGE
    timeout: 90
    incompatible_lists: [motion_behaviors]
    capabilities: [VISUAL_SERVOING]

  - behavior: PAY_ATTENTION_TO_VISUAL_MARKERS
    recurrent: yes
    capabilities: [VISUAL_MARKERS_RECOGNITION]

  - behavior: KEEP_HOVERING
    category: recurrent
    incompatible_lists: [motion_behaviors]
    capabilities: [SETPOINT_BASED_FLIGHT_CONTROL]


  - behavior: WAIT

    arguments:
      - argument: DURATION
```

```
        allowed_values: [1,1000]
      - argument: UNTIL_OBSERVED_VISUAL_MARKER
        allowed_values: [0,1023]

  - behavior: FLIP
    incompatible_lists: [motion_behaviors]
    capabilities: [SETPOINT_BASED_FLIGHT_CONTROL]
    arguments:
      - argument: DIRECTION
        allowed_values: [BACK, FRONT, LEFT, RIGHT]



  - behavior: SELF_LOCALIZE_BY_ODOMETRY
    category: recurrent
    incompatible_lists: [self_localization_behaviors]
    capabilities:
      - SELF_LOCALIZATION_BY_ODOMETRY
      - DYNAMIC_SELF_LOCALIZATiON_MODE
    ROS_service_calls: [change_self_localization_mode_to_odometry]

  - behavior: SELF_LOCALIZE_BY_VISUAL_MARKERS
    category: recurrent
    incompatible_lists: [self_localization_behaviors]
    capabilities:
     -  SELF_LOCALIZATION_BY_ODOMETRY
      - VISUAL_MARKERS_RECOGNITION
      - SELF_LOCALIZATION_BY_VISUAL_MARKERS
      - DYNAMIC_SELF_LOCALIZATiON_MODE
    ROS_service_calls: [change_self_localization_mode_to_visual_markers]

  - behavior: SLAM_BY_VISUAL_MARKERS
    category: recurrent
    incompatible_lists: [self_localization_behaviors]
    capabilities:
     -  SELF_LOCALIZATION_BY_ODOMETRY
      - VISUAL_MARKERS_RECOGNITION
      - SELF_LOCALIZATION_BY_VISUAL_MARKERS
      - OBSTACLE_DETECTION_BY_VISUAL_MARKERS
      - DYNAMIC_SELF_LOCALIZATiON_MODE

  - behavior: BROADCAST_MESSAGE
    arguments:
      - argument: TEXT
        allowed_values: TEXT

  - behavior: GENERATE_PATH_FREE_OF_OBSTACLES
    category: deliberative
    capabilities: [MISSION_PLANNING]
    arguments:
      - argument: DESTINATION
        allowed_values: [-100, 100]
        dimension: 3

behavior_lists:
  - list: self_localization_behaviors
    behaviors:
      - SELF_LOCALIZE_BY_ODOMETRY
      - SELF_LOCALIZE_BY_VISUAL_MARKERS
      - SLAM_BY_VISUAL_MARKERS
  - list: motion_behaviors
    behaviors:
      - TAKE_OFF
      - LAND
      - FLIP
      - KEEP_HOVERING
      - FOLLOW_OBJECT_IMAGE
      - START_MOVING
      - START_HOVERING
      - GO_TO_POINT
      - ROTATE

reactive_activation:
```

```
  - behavior: KEEP_HOVERING
    condition: flight_state(self, FLYING)
    priority: lower

  - behavior: LAND
    condition: charge(battery, ?X), less_than(?X, 10)
    priority: higher

  - behavior: SELF_LOCALIZE_BY_ODOMETRY
    priority: lower


capability_descriptors:

  - capability: SETPOINT_BASED_FLIGHT_CONTROL
    process_sequence: [droneTrajectoryController]
    incompatible_capabilities: [VISUAL_SERVOING]

  - capability: PATH_PLANNING
    process_sequence: [droneTrajectoryPlanner, droneYawPlanner]

  - capability: VISUAL_SERVOING
    process_sequence: [trackerEye, open_tld_translator, droneIBVSController]
    incompatible_capabilities: [SETPOINT_BASED_FLIGHT_CONTROL]

  - capability: DYNAMIC_SELF_LOCALIZATION_MODE
    process_sequence: [self_localization_mode_selector]

  - capability: SELF_LOCALIZATION_BY_ODOMETRY
    permanent_active: yes
    process_sequence: [droneOdometryStateEstimator]

  - capability: SELF_LOCALIZATION_BY_VISUAL_MARKERS
    process_sequence:
      - droneLocalizer

  - capability: OBSTACLE_DETECTION_BY_VISUAL_MARKERS
    process_sequence:
      - droneObstacleDistanceCalculator
      - droneObstacleProcesser

  - capability: VISUAL_MARKERS_RECOGNITION
    process_sequence:
      - droneArucoEyeROSModule
```