Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

An execution control method for the Aerostack aerial robotics framework*

Martin MOLINA†‡1, Alberto CAMPORREDONDO1, Hriday BAVLE2,
Alejandro RODRIGUEZ-RAMOS2, Pascual CAMPOY2

1Department of Artificial Intelligence, Universidad Politécnica de Madrid, Madrid 28040, Spain
2Centre for Automation and Robotics, Universidad Politécnica de Madrid, Madrid 28040, Spain
†E-mail: martin.molina@upm.es

Received Sept. 11, 2018; Revision accepted Nov. 27, 2018; Crosschecked Jan. 8, 2019

Abstract: Execution control is a critical task of robot architectures which has a deep impact on the quality of the final system. In this study, we describe a general method for execution control, which is a part of the Aerostack software framework for aerial robotics, and present technical challenges for execution control and design decisions to develop the method. The proposed method has an original design combining a distributed approach for execution control of behaviors (such as situation checking and performance monitoring) and centralizes coordination to ensure consistency of the concurrent execution. We conduct experiments to evaluate the method. The experimental results show that the method is general and usable with acceptable development efforts to efficiently work on different types of aerial missions. The method is supported by standards based on a robot operating system (ROS) contributing to its general use, and an open-source project is integrated in the Aerostack framework. Therefore, its technical details are fully accessible to developers and freely available to be used in the development of new aerial robotic systems.

Key words: Aerial robotics; Control architecture; Behavior-based control; Executive system
https://doi.org/10.1631/FITEE.1800552 **CLC number:** TP242.6

1 Introduction

Execution control is a common task in robot control architectures to communicate two description levels: a level where a requester, such as a human operator or an automatic decision system, tells a robot what to do using symbolic commands, and a level where a number of computational processes concurrently run to generate the required functionalities. Execution control verifies that the requested command is correct before execution, ensures a consistent and efficient execution of concurrent processes during execution, and communicates results in terms of success or failure after execution.

Practical experience in the development of robot architectures shows that the methods used for execution control are critical and have a deep impact on the quality of the final system, such as the safety and complexity of human-robot interaction. Kortenkamp et al. (2008) pointed out that designing robotic architectures is much more of an art than a science. Therefore, there is a need for methods that reduce the required effort to build new system architectures.

We present a general method for execution control to develop an existing open-source software framework called "Aerostack" (Sanchez-Lopez et al., 2017). The method is designed for the framework, considering the requirements to facilitate the development of new systems.

‡Corresponding author

* Project supported by the European Union's Horizon 2020 Research and Innovation Program under the Project ROSIN (No. 732287)

ORCID: Martin MOLINA, http://orcid.org/0000-0001-7145-1974

2 Related work

A three-layer architecture is a popular control architecture for autonomous robots (Bonasso et al. 1997), which has different versions, such as Atlantis (Gat, 1992) and LAAS (Alami et al., 1998). This architecture uses three layers (Kortenkamp et al., 2008) as follows:

1. Behavioral control layer or functional layer is the lowest level and consists of behaviors that carry out actions of a robot. Each behavior connects with sensors perceiving environments to actuators, thus creating a sensor-action loop.

2. Executive layer is an intermediate layer that activates and deactivates behaviors to achieve high-level tasks, thus avoiding conflicts among behaviors using the same actuators.

3. Planning layer or decision layer is the highest level and is responsible for determining the long-range activities of a robot based on mission goals.

The executive layer includes a task decomposition module with a specialized language to help developer specify how to execute a task considering different situations in the environment of both regular situations and contingencies. The languages used are reactive action packages (RAPs) (Firby, 1989), execution support language (ESL) (Gat, 1996), and plan execution interchange language (PLEXIL) (Verma et al., 2005). The executive layer also includes an additional component that serves as a bridge with the behavioral control layer. The additional component can be called “execution control system” in the LAAS architecture (Alami et al., 1998), which includes an execution control system called “Kheops” using production rules. Request and resource checker (R2C) (Ingrand and Py, 2002) is another execution control system in the LAAS architecture which is oriented to fault protection and tries to solve some limitations of Kheops. Rutten (2001) proposed an execution control system focusing on model checking.

We present a new method for execution control, which is a part of the Aerostack aerial robotics software framework. Compared with other work, the new method is designed to satisfy special requirements of Aerostack (see details in Section 4) which aim at facilitating the construction of real robotic systems and emphasizing utility aspects in robot architectures, such as timeliness in the development and performance of effectiveness (Arkin, 1998).

3 Aerostack software framework

Modern autonomous aerial systems integrate multiple computational methods, such as computer vision algorithms, motion controllers, self-localization and mapping methods, automated planning, and coordination algorithms. Integration of such heterogeneous methods in a complex operational system is a technical challenge that requires efficient and robust architectural solutions.

We have developed the Aerostack software framework (www.aerostack.org) in our group Computer Vision and Aerial Robotics (CVAR) at Universidad Politécnica de Madrid (UPM) (Sanchez-Lopez et al., 2017). This framework based on a robot operating system (ROS) provides a library of software components specialized in aerial robotics and a general combination mechanism using an architectural pattern that guides the integration process.

The framework in our group contributes to a great productivity in the construction of new systems and experiments with new algorithms. For example, Aerostack has been used with great success by our group in the International Aerial Robotics Competitions, such as IMAV 2013, 2016, and 2017 and IARC 2014. Aerostack has also been used by CVAR in complex robotic systems related to natural user interfaces (Suárez Fernández et al., 2016), surface inspection (Molina et al., 2018), coordinated multi-robot systems (Sampedro et al. 2016), landing on moving platforms (Rodriguez-Ramos et al., 2017), search-and-rescue missions (Sampedro et al. 2018), and altitude estimation in complex dynamic environments (Bavle et al., 2018).

Fig. 1 shows the main components of the Aerostack architecture (version 3.0), which shares the behavioral control layer (or behavioral layer in short) and executive layer of the three-layer architecture. In addition, the architecture uses a communication layer to communicate interfaces.

The behavioral layer in Aerostack includes components providing basic functionalities of an aerial robotic system, which are as follows:

1. Feature extractors that read simple states of sensors and implement complex vision and pattern recognition algorithms; for example, a feature extractor is a recognizer of visual markers (e.g., ArUco markers or QR codes).

2. Motion controllers that implement combinations of proportional-integral-derivative (PID) controllers. For example, motion controllers can be used for speed control and height control.

3. SLAM processes that perform self-localization and mapping (SLAM) using, for example, extended Kalman filter (EKF) techniques.

4. Motion planners that generate obstacle-free paths to reach destination points.

5. Methods that communicate with other agents for robot collaboration or human collaboration.

Aerostack provides a software library of these components. A developer can configure specific components from the library and combine them to build a particular robotic system architecture. In this architecture, the basic executable component is a process, which is defined according to a specific function (e.g., a path planner, an obstacle recognizer, or a speed controller). Each process in Aerostack is implemented as an ROS node. Aerostack concurrently runs processes that communicate with each other using message passing methods for inter-process communication provided by an ROS: a request-reply service (an ROS service) and a publish-subscribe method filtering messages using a topic-based approach (an ROS topic).

The executive layer in the Aerostack consists of three systems: mission control system, execution control system, and belief management system. The mission control system receives an input as a mission plan specified in certain formal language, such as task-based mission specification language (TML) in the Aerostack (Molina et al., 2017), but a standard language (such as Python) or a graphical editor with behavior trees can also be used. The mission control system verifies the correctness of plan and interprets it to sequentially generate execution requests, such as activations and deactivations of robot behaviors.

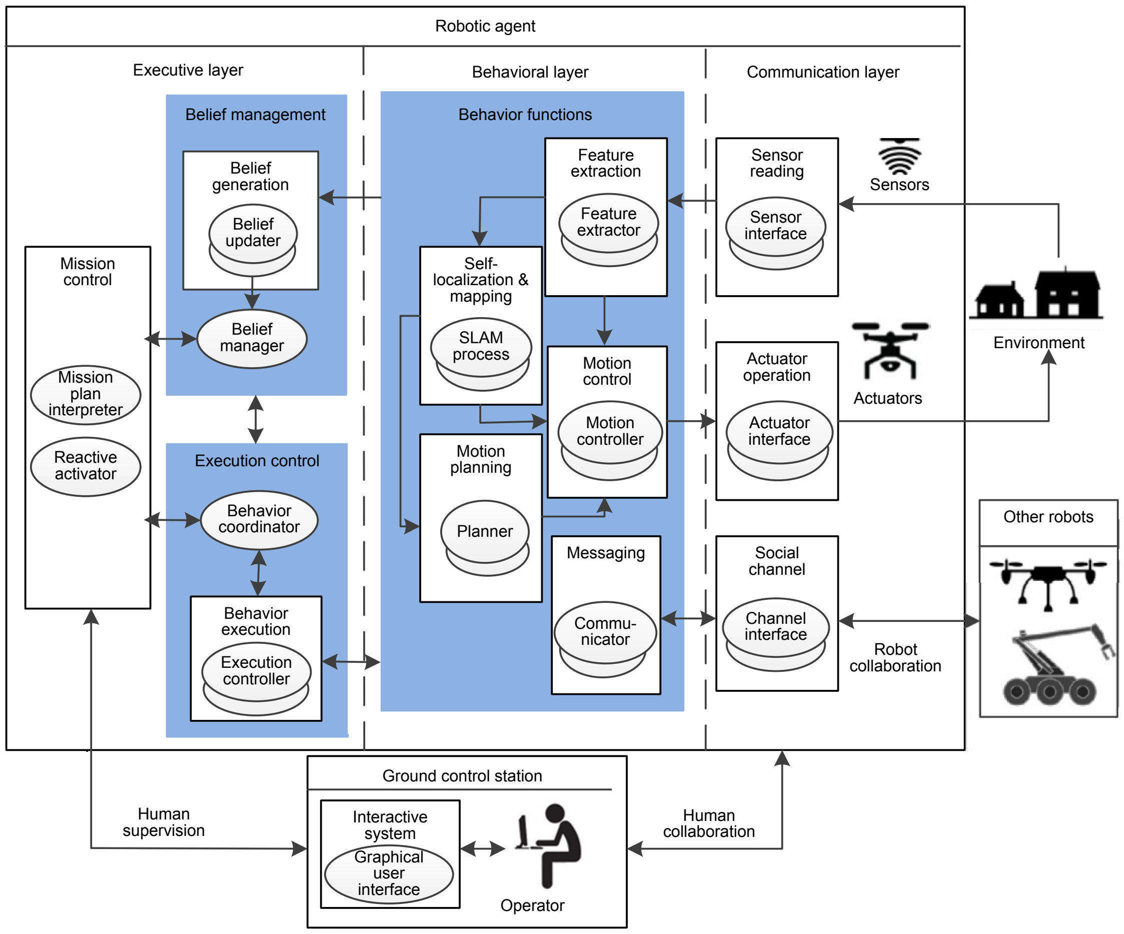

Fig. 1 Aerostack architecture (version 3.0)

The execution control system translates these requests into orders to start or stop processes to reconfigure the set of concurrent behavioral layer processes. The belief management system maintains a consistent memory of relevant data about the world, which is necessary for decisions related to mission planning. In the next section, we will describe the details of these two systems.

4 Requirements for execution control

The execution control system establishes a clean separation among decisions about mission tasks and realizes these tasks with concurrent processes. The execution control system creates a logical interface in Aerostack helping a developer specify mission plans in a simpler way and verifies that the mission plan is executed as expected.

By the specification of mission plans, the execution control system relieves the developer from specifying excessive details about how to control the execution of a task.

By the execution of mission plans, the execution control system verifies that each task is consistent with the environmental situation, thus preventing a robot from performing wrong or dangerous behaviors. The execution control system reports the presence of unexpected events, which can be used by a decision system to formulate a more robust plan or by a human operator to supervise the correct execution of the mission plans.

Fig. 2 shows the role of the execution control system. In a general way, a requester is used to designate a subject who decides what to do next and asks a robot to do it. A human operator could be a requester or, in a multi-layered architecture of an autonomous system, the requester can be an automatic system (e.g., the mission control system in Aerostack).

The execution control system analyzes the feasibility of an execution request to accept or reject it before it is executed, and responds to each request by reconfiguring the set of running processes. In addition, to help a requester make the next decision, the execution control system communicates results of the execution by notifying the results in terms of success or failure and relevant changes in the environment.

One of the important requirements of the execution control system is to use an appropriate representation for the requester to express what to do and interpret execution results at an adequate level of abstraction. This representation should be expressive enough, but at the same time, it should abstract execution details which are not necessary to make decisions about mission tasks to the requester.

There are two additional requirements derived from Aerostack's general goal of facilitating the development of robotic systems:

1. Generalization. The solution must be applicable in the construction of different aerial robotic systems and missions.

2. Usability. The solution must be useful for a developer in building the execution control system of a specific robot architecture, with an acceptable effort in the context of a development project.

Finally, there are two requirements related to the integration in Aerostack:

1. Using standards. The solution must use programming standards of the robotic systems that are used by Aerostack (e.g., ROS middleware).

2. Accessibility. The solution should be freely accessible as a part of an open-source framework.

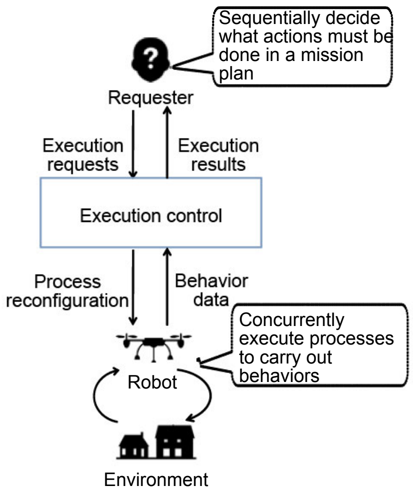
Sequentially decide what actions must be done in a mission plan
Requester
Execution requests
Execution results
Execution control
Process reconfiguration
Behavior data
Concurrently execute processes to carry out behaviors
Robot
Environment

Fig. 2 Role of the execution control system

5 Representation for the execution requester

In this section, we describe the representation designed by a requester to operate with the execution control system, concerning an adequate level of abstraction.

5.1 Representation of execution requests

The execution control system uses behaviors to express what actions a robot must do. Behavior is a natural notion which is familiar to general users; in simple words, a behavior is anything a robot can do: taking off, moving forward, landing, etc. It has been traditionally used in robotics for building systems (Brooks, 1986; Arkin, 1998) and as a basic concept in specifying mission plans (Rothenstein, 2002; Allgeuer and Behnke, 2013). In robotic systems, a behavior integrates perception algorithms and actuation controllers to generate a particular pattern perception-actuation. This integration helps hide details remaining encapsulated about perception-action. Therefore, a behavior is useful to abstract implementation details and express, in simple words, how the robot must operate.

5.1.1 Behavior activation requests

The execution control system uses behavior activations to formulate requests. Activation a_i by requester r_j of behavior b_k can be formally represented as

$$a_i=<r_j, b_k, \{<p_1, v_1>, <p_2, v_2>, \ldots, <p_n, v_n>\}>$$

where $\{<p_1, v_1>, <p_2, v_2>, \ldots, <p_n, v_n>\}$ is a set of parameter value pairs that configure the behavior activation. For example, to activate the behaviors of an aerial robot—taking off, rotating 45°, and going to a point with coordinate (1.0, 2.5, 7.0)—the following activations can be used (where R1 is the requester):

a_1=<R1, TAKE_OFF, {}>
a_2=<R1, ROTATE, {<angle, 45°>}>
a_3=<R1, GO_TO_POINT,
{<coordinate, (1.0, 2.5, 7.0)>}>

In practice, for readability, this is actually written in the following way (asked by the requester R1):

```
TAKE_OFF
ROTATE angle: 45°
GO_TO_POINT coordinate: (1.0, 2.5, 7.0)
```

A requester can ask an activation a_i to initiate or terminate. When an initiation of the activation is requested, the execution control system confirms that it has been accepted or, on the contrary, rejected because of certain reasons.

5.1.2 A library of behaviors

One of the practical utilities of a tool, such as Aerostack, is to provide a powerful and versatile set of behaviors for aerial robotics that can be used by a requester as a part of its language for execution. A library of behaviors was designed for Aerostack, which considers quality principles for each behavior (e.g., generality, stability, clarity, and conciseness). A part of this library is fully implemented in Aerostack and used in this study.

In the library, there are goal-based and recurrent behaviors. Goal-based behaviors are defined to reach a final state (attaining a goal). Examples of these behaviors are flight maneuvers related to rotors, such as simple flight maneuvers (TAKE_OFF, LAND, KEEP_HOVERING, KEEP_MOVING, and ROTATE) and more complex flight maneuvers such as moving avoiding obstacles (GO_TO_POINT) and following objects (FOLLOW_OBJECT_IMAGE). The category of goal-based behaviors may include behaviors related to other effectors, such as light (TURN_LIGHT), camera (TAKE_PHOTO and TAKE_VIDEO), dropping mechanism (DROP_ITEM), moving camera (MOVE_CAMERA), and sound (SAY_SENTENCE).

Recurrent behaviors recurrently perform an activity or maintain a desired state. Self-localization behaviors belong to this category (SELF_LOCALIZE_BY_LIDAR, SELF_LOCALI ZE_BY_VISUAL_MARKERS, etc.). Other examples are attention (PAY_ATTENTION_TO_VOICE_COMMANDS, PAY_ATTENTION_TO_COLORS, etc.), communications (SPEAK_UP, etc.), data storage behaviors (BUILD_MAP and RECORD_VIDEO), etc.

5.2 Representation of execution results

Activation of behaviors usually produces a number of effects. One of the functions of the execution control system is to inform the requester about these effects at an adequate level of abstraction. This is done with two classes of information: the cause of behavior termination and a memory of beliefs about the world.

5.2.1 Cause of behavior termination

After the activation of a behavior, a message may be generated by the execution control system to inform a requester about the cause of its termination. The possible values of this message are:

1. goal achieved: the behavior has achieved its goal;
2. time out: the behavior has not achieved its goal in the expected time;
3. wrong progress: the behavior has not progressed as expected;
4. interrupted: the behavior execution has been cancelled.

This information is useful for a requester to confirm that a behavior has operated as expected (therefore, the mission can continue as planned) or, on the contrary, the behavior has failed (therefore, the next step must be reconsidered); for example, execution failures happen because the requester may have decided to activate a behavior based on assumptions about the environment, which are not satisfied when the behavior is executed (which is usual with autonomous robots operating in the uncertain and dynamic environment).

5.2.2 Memory of beliefs about the world

Another type of information which is useful to a requester is how the world may have changed because of the operation of behaviors. Therefore, the execution control system may operate together with a memory of beliefs that abstract relevant data about the world. This memory acts as a filter that presents the requester, in a uniform way, the necessary information for decisions related to mission plans.

The memory uses a particular representation for beliefs. A belief is a proposition about the world that the robot believes is true (the “world” here refers to both the external world and the internal state of a robot).

Beliefs are represented using a logic-based approach with predicates. Examples of the predicates with a general format of predicate(object, value) and the simple form property(object) are shown in Table 1. The representation follows an object-oriented approach. Objects are represented with numerical identifiers as instances of a class; for example, object(32, obstacle) represents that object 32 is an obstacle. Additional predicates are used to represent attributes of objects; for example, color(32, blue) represents that object 32 is blue.

Attribute values defined for an object are assumed to be mutually exclusive; for example, the belief charge(92, empty) is incompatible with the belief charge(92, full), because the empty and full values are mutually exclusive (the identifier of 92 represents the battery). If a particular predicate has values that are not mutually exclusive, this must be treated as an exception. The predicate visible(x) is used to indicate that an object is currently observed.

Table 1 Examples of predicates representing beliefs

Predicate	Description
Object(x, y)	Object x is an instance of class y
Position(x, y)	Object x is at the position y
Name(x, y)	Name of object x is y
Flight_state(x, y)	Aerial robot x is in flight state y
Code(x, y)	Numerical code of x is y
Color(x, y)	Color of x is y
Charge(x, y)	Charge of x is y
Carry(x, y)	Agent x carries object y
Visible(x)	Object x can be observed

This representation can be used by a robot to describe the requester how the world changes, and the requester can use this representation to ask the execution control system to memorize details about a mission execution which may be useful for future decisions. For example, a requester can ask the execution control system to memorize the coordinates of the current place with the name point A and with a set of beliefs as

object(57, place)
name(57, point_A)
position(57, (3.0, 4.5, 0.0))

6 Execution control method

In this section, we describe the distributed approach of the execution control system designed for Aerostack and the architecture of processes that implement the execution control system.

6.1 Distributed approach

Fig. 3 shows the distributed scheme used by the method of the execution control system. A network of execution processes, i.e., the processes of a behavioral layer of the Aerostack architecture (such as processes for the perception and motion control system), which can run or stop in a particular moment, is shown at the bottom of Fig. 3.

The method of the execution control system responds to each request by reconfiguring the set of execution processes that are running. The consistency is ensured by checking conditions which are distributed in different processes. Fig. 3 shows the conditions used and how they are distributed in processes.

6.1.1 Behavior-based distribution

For each behavior, there is a behavior execution controller, which is a separate process (in some cases, for example, when there is only one execution process for a behavior, it may be more efficient to have a single process that integrates the behavior execution controller with the execution process) that starts or stops the execution processes of behavior and performs local consistency checking using two types of information: situation conditions and performance conditions.

The verification of situation conditions is done before a behavior is executed and is important due to the property of situated behaviors, meaning that each behavior works in only specific situations; e.g., to execute the behavior LAND, the robot must be flying. In Aerostack, instead of using a model-based mechanism for this verification (e.g., with finite state machines or Petri nets), situation conditions are directly verified using data about the state of the environment; e.g., data derived from sensors is present in the belief memory or in the content of messages of execution processes.

Situation conditions are distributed in different execution controllers. Conditions of each controller must be formulated with enough precision to correctly classify the required situation for the behaviors. The precision depends on the representation used for conditions, and on the capacity of a robot to perceive the environment from sensors.

Performance conditions express the expected performance of a particular behavior, e.g., the maximum expected time to achieve a goal. The verification of these conditions is important for monitoring the execution and detecting success or failure. The

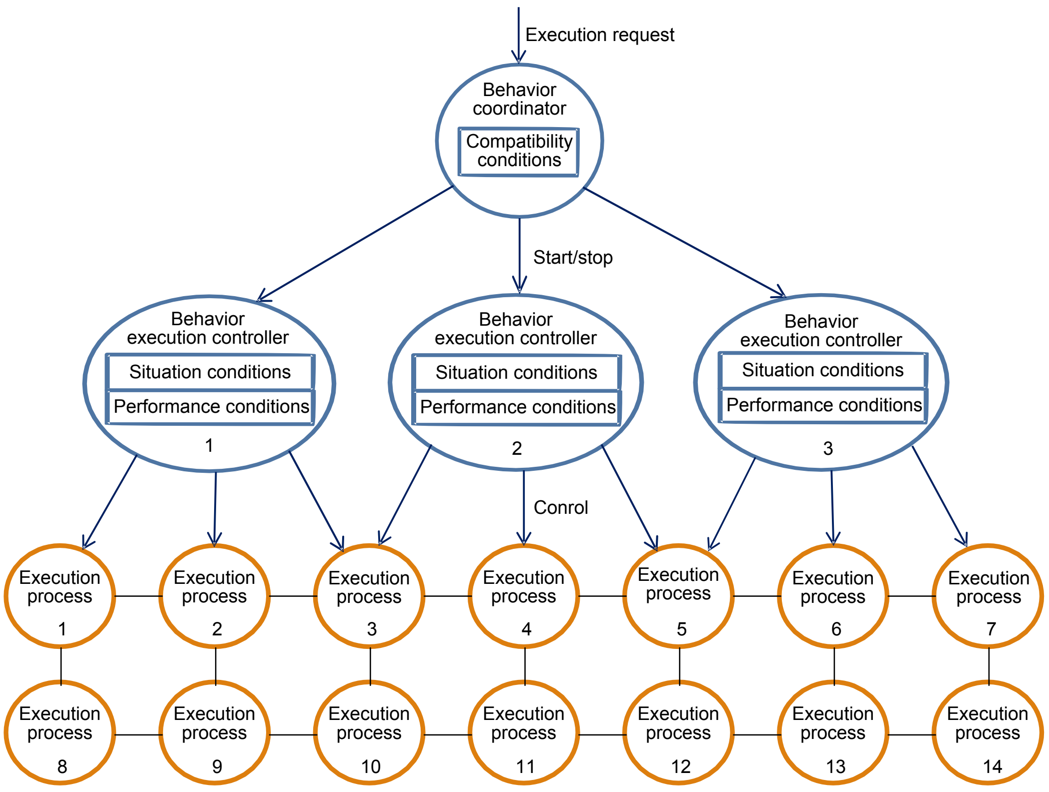

Fig. 3 Distributed scheme for execution control

verification is periodically done using data about the environmental states. This monitoring provides cognizant failures (Gat, 1996, 1998), i.e., the ability of a system to detect failures, which is important for a robot to operate in an unpredictable environment.

Performance conditions are written for each behavior and should be able to identify as many success or failure situations as necessary for correct decision making during the mission plan execution.

The modularity provided by execution controllers, which separate and encapsulate the execution details of each behavior, helps add new behaviors with flexibility, without affecting other behaviors and the overall execution control mechanism.

6.1.2 Central coordination

The method of execution control system includes a central process called "behavior coordinator" that ensures the global consistency of a set of active behaviors. The coordinator performs a kind of configuration task (Mittal and Frayman, 1989) and responds to the requests of behavior activation or inhibition by configuring the set of active behaviors, thus ensuring that a set of compatibility conditions are satisfied.

The compatibility conditions are explicitly written for a particular control architecture that uses a set of behaviors. The compatibility conditions are expressed at the level of behaviors, but they are based on properties at the level of execution processes. There is a type of condition based on Definition 1.

Definition 1 (Incompatible behaviors) Two behaviors b_1 and b_2 are incompatible, noted as incompatible(b_1, b_2), if they cannot be executed at the same time, because there is at least an execution process p used by behaviors b_1 and b_2 but with different input data, or they produce contradictory effects in the environment; i.e., effects of behaviors b_1 and b_2 in the environment are mutually opposed.

A condition related to this type of compatibility is expressed by a set of behaviors $B=\{b_1, b_2, ..., b_n\}$ that are mutually exclusive; i.e., every pair of behaviors in B, b_i and b_j, satisfies the relation incompatible(b_i, b_j). For example, there may be a condition expressing that a set of behaviors corresponding to flight maneuvers (TAKE_OFF, LAND, KEEP_HOVERING, etc.) are mutually exclusive. The reason for this constraint is that these behaviors use common processes for the motion control system with different input data.

There is another type of condition expressing precedence between behaviors b_1 and b_2. This condition is based on Definition 2.

Definition 2 (Precedence relation) There is a precedence relation between behaviors b_1 and b_2, noted as preceding(b_1, b_2), if behavior b_1 must be active before b_2, because the execution processes of b_1 generate output values required as inputs for the execution processes of b_2.

The coordinator includes a precedence condition for every pair of behaviors b_1 and b_2, which satisfies preceding(b_1, b_2). The precedence conditions set must not present loops in the paths established by pairs of behaviors.

Note that these conditions are specific to the set of behaviors used in a particular robotic system, and that they are manually written by a developer. In principle, it would be desirable to automatically generate conditions from the structure of execution processes, which theoretically seems possible except for behaviors producing contradictory effects in the environment. However, in the current implementation, we find that this would add more complexity to the development of new behaviors, requiring more efforts and being more error-prone compared with manually writing conditions. In fact, because conditions are defined at the level of behaviors instead of the execution processes, the representation is simple and therefore easy to maintain.

The coordinator follows a priority scheme to activate behaviors based on Definition 3.

Definition 3 (Activation priority) Activation a_1 has a higher priority than activation a_2, noted as priority(a_1, a_2), if the requester of a_1 has a higher priority than the requester of a_2.

In our current implementation, there are two requesters: an emergency requester requesting the activation when an emergency is detected following a reactive approach and a mission plan requester requesting activations step by step, to carry out a mission following a deliberative approach. There are default requests that correspond to default behaviors. Default behaviors are behaviors that keep operating the basic functions of a robot.

For example, an aerial robot should, by default, activate a behavior to hover in the air when it is flying, when no other behaviors related to flight maneuvers are active. The following priority order is assumed in our current implementation: emergency requester > mission plan requester > default.

6.2 Detailed architecture

Fig. 4 shows the detailed architecture of the execution control system corresponding to the actual software implementation. However, for clarity, some services and topics have been presented in the study with different names; e.g., the service request_ behavior_activation is implemented with the name of activate_behavior.

There are five components in Fig. 4: three specific processes (e.g., behavior coordinator, belief manager, and process manager) and two classes of processes (behavior execution controller and belief updater) instantiated into specific processes; for example, the class of behavior execution controller process is instantiated into the behavior GO_TO_ POINT process. In the following subsections, we will describe these five components in detail.

6.2.1 Behavior execution controllers

As described above, there is a separate process called "behavior execution controller" for each behavior. Each execution controller is implemented as an ROS node that provides the request-reply services:

1. Checking activation conditions. This service verifies that the behavior can be activated because environment is consistent with its situation conditions.

2. Starting behavior. This service starts the execution in the following way: It launches the execution processes of the behavior. Then, it publishes messages with the convenient parameter values for these processes. Finally, the execution controller monitors the performance of the execution using the performance conditions.

3. Stopping behavior. This service stops the behavior by requesting that the execution processes of the behavior are stopped.

For example, there may be an execution controller for the behavior GO_TO_POINT, which may use two processes: motion controller for motion control and trajectory planner to generate trajectories that are free of obstacles.

To start this behavior, the execution controller launches the motion controller and trajectory planner processes by sending messages to the process manager, and then publishes a message with the destination point (a parameter given as an input). This message is used by the trajectory planner process to generate a candidate trajectory, which is used by the motion controller process to move the robot.

Finally, the execution controller monitors messages related to the robot position published by processes related to self-localization and mapping to detect if the robot reaches its destination. Because of this performance of monitoring, the behavior execution controller may publish a message (for the ROS topic end-of-behavior) to communicate the cause of termination with the values related to success or failure (such as goal achieved, time out, wrong progress, or interrupted).

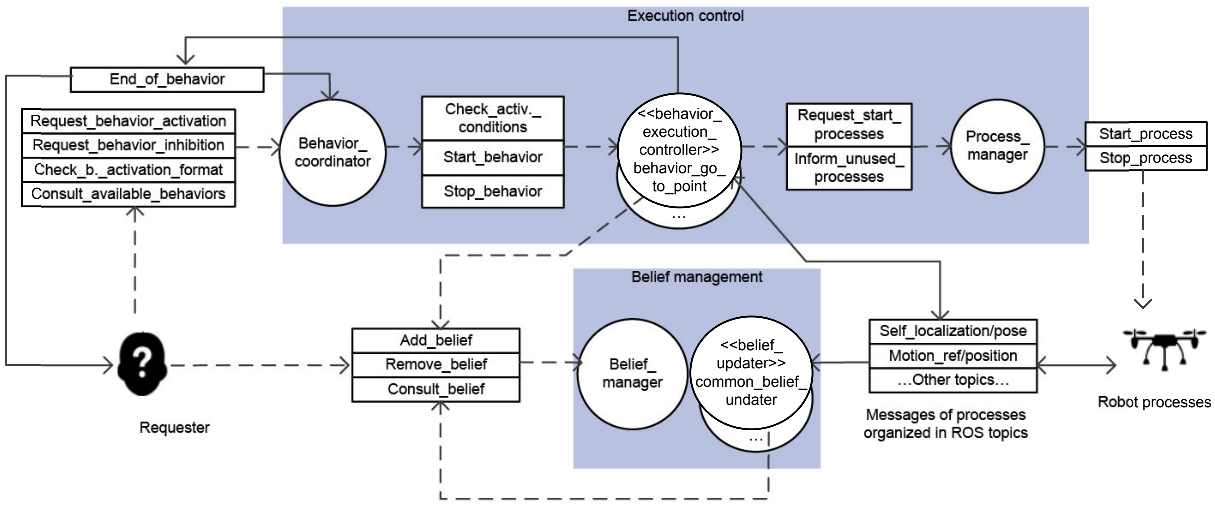

Fig. 4 Detailed architecture of the execution control system

Circles: Aerostack processes (ROS nodes); rectangles with dashed arrows: request-reply ROS services; rectangles with continuous arrows: publish-subscribe messages as ROS topics

6.2.2 Behavior coordinator

The behavior coordinator is a process that responds to behavior activation or inhibition requests by reconfiguring a set of active behaviors.

The behavior coordinator uses a catalog which is an information resource containing metadata about behaviors. The catalog is written by a developer and stored in a file using YAML format.

Fig. 5 shows a partial example of the catalog that includes a list of behavior descriptors. Each descriptor specifies information about the behaviors, such as name, category (goal-based or recurrent), time out (in seconds), default (the value of "yes" means that this behavior must be active, unless another incompatible behavior is already active), processes (a list of execution processes that are required to execute the behaviors), and parameters (with names and allowed values).

```
behavior_descriptors:

- behavior: GO_TO_POINT
   category: goal_based
   timeout: 120
   default: no
   processes:
     - droneTrajectoryController
     - droneTrajectoryPlanner
     - droneYawPlanner
   parameters:
     - parameter: coordinates
       allowed_values: [-100,100]
       dimensions: 3
...

compatibility_conditions:

- mutually_exclusive:
  - TAKE_OFF
  - LAND
  - KEEP_HOVERING
  - GO_TO_POINT
  - ROTATE
  ...

- active:
    - SELF_LOCALIZE_BY_VISUAL_MARKERS
  before:
    - GO_TO_POINT
    - ROTATE
    - KEEP_HOVERING
    ...

 ...
```

Fig. 5 A partial example of behavior catalog

The catalog includes a list of compatibility conditions that are used for consistency checking of concurrent execution (Fig. 5); for example, there is a condition expressing that a set of behaviors, such as taking off and landing, corresponding to flight maneuvers, are mutually exclusive, and a precedence constraint ensuring that a behavior related to self-localization must be active before the activation of certain motion behaviors.

In the design of a language for catalog, we pay special attention to keeping it simple and flexible and making it easy for developers to add new behaviors. The catalog uses a declarative representation. Each behavior descriptor is independent of others, and conditions are independent of each other; thus, they can be added or removed with flexibility.

The coordinator is implemented as an ROS node and provides the following two request-reply services related to behavior activation:

1. Requesting behavior activation. This service requests the activation of a behavior and can be accepted or rejected.

2. Requesting behavior inhibition. This service stops an active behavior. Therefore, it uses the service of stop behavior of the execution controller.

The service of requesting behavior activation analyzes whether the request satisfies the activation conditions of the corresponding behavior. If the conditions are not satisfied, the service rejects the activation; otherwise, the algorithm checks whether the behavior is compatible with other behaviors that are already active. If an incompatible behavior with a high priority is active, then this service rejects the request.

If there are incompatible behaviors that are active with a low priority, the service will stop them, but these stops may generate the activation of other behaviors by default. These potential activations are verified in advance to guarantee that they satisfy the precedence constraints. Finally, when all the conditions are satisfied, this service stops the incompatible behaviors and starts the necessary behaviors.

The coordinator receives a message when a behavior terminates with success or failure (topic end-of-behavior). When the coordinator receives this message, it executes a procedure removing the behavior from the list of active behaviors and activating all default behaviors, which are incompatible with the removed behavior. The activation of default behaviors verifies the precedence, and compatibility constraints guarantee that they are satisfied.

Fig. 6 shows an example of behavior coordination to illustrate how the coordinator works. Two

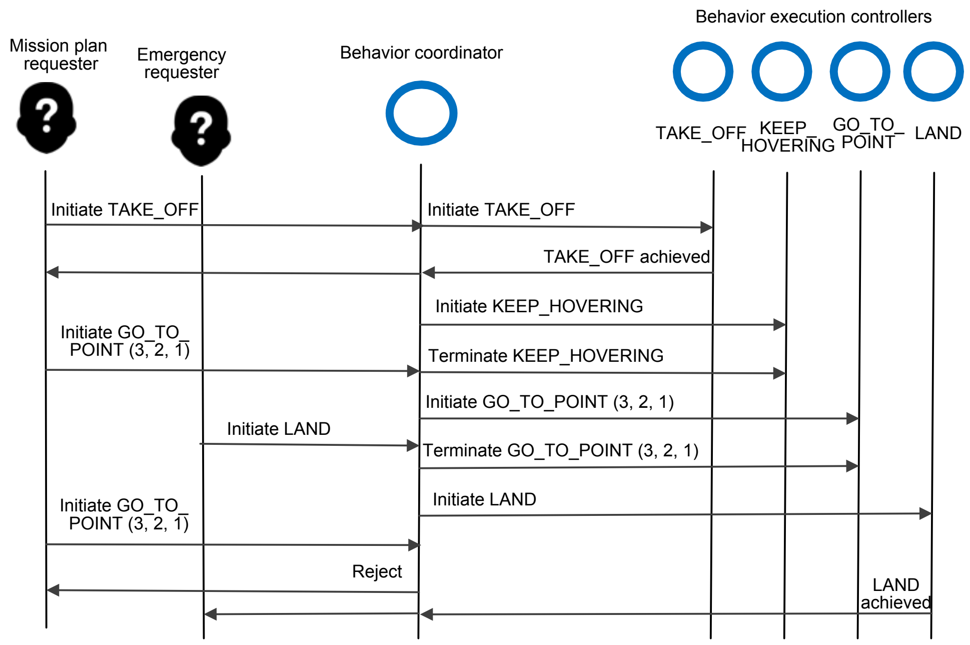

Fig. 6 Example of behavior coordination

requesters (mission plan requester and emergency requester) send activation requests to the coordinator, and thus the coordinator sends requests to the execution controllers. Fig. 6 shows how the coordinator activates and terminates a default behavior that is not requested by both requesters (behavior KEEP_HOVERING) and how the coordinator rejects the activation from the first requester when another behavior from the second requester with a high priority is active.

Fig. 7 shows an extension of the architecture that is necessary to facilitate human-robot collaboration. The idea of this extension is that a behavior execution controller can request the activation or the inhibition of other behaviors; for example, it is useful to have behavior REQUEST_OPERATOR_ASSISTANCE, which can pause the execution of certain behaviors while interacting with an operator. Details of this extension can be found in Molina et al. (2018).

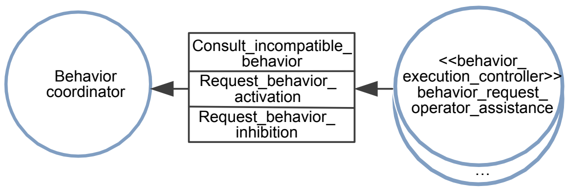
Behavior coordinator
Consult_incompatible_ behavior
Request_behavior_ activation
Request_behavior_ inhibition
<<behavior_ execution_controller>> behavior_request_ operator_assistance
...

Fig. 7 Extension of the architecture to facilitate human-robot collaboration

Besides the request-reply services, the coordinator provides the service of checking behavior activation format. This service checks that a behavior activation expression (a text sentence) has a correct format with a valid behavior name, arguments, and values for arguments using the behavior descriptors of the catalog. This validation can protect the system against producing errors when a request is written by a human operator.

6.2.3 Process manager

The process manager starts and stops the execution of the processes by efficiently using the resources. When a behavior is activated, its execution controller asks the process manager to start the execution processes. When the behavior is inhibited, its execution controller informs the process manager that the processes are not needed and then the process manager stops unnecessary processes to minimize resource consumption. Therefore, the process manager provides two request-reply services: requesting start processes and informing unused processes.

Because an execution process p_i may be used at the same time by different behaviors, process p_i is stopped only when it is not used by any behavior. Therefore, the process manager maintains an updated reference counter r_i for each process p_i to count the number of behaviors using process p_i in a given moment. Only when the number of references is zero (e.g., r_i=0), does the process manager stop process p_i.

To avoid an unnecessary stop when two consecutive behavior activations use the same process, the process is not immediately stopped when the service of informing unused process is called. It is delayed to keep the process running after the service of requesting start processes is called. The service of requesting start processes updates the reference counters with new values, and then the processes with zero references are stopped. This strategy is useful to provide the continuity to the process execution and avoid inefficient disruptions.

6.2.4 Belief manager

The belief manager stores the sets of beliefs, maintains their consistency, and provides the request-reply services of adding belief, removing belief, and consulting belief. For the services of adding belief and removing belief, the belief manager updates the content of the memory of beliefs and maintains consistency among beliefs which are mutually exclusive. When a belief is added, e.g., charge(92, empty), the incompatible beliefs are automatically retracted, e.g., charge(92, full).

The service of consulting belief can be used to know whether a belief is true and determine parameter values, using belief expressions that may include variables. For example, a consultation with the expression

object(?*x*, battery), charge(?*x*, ?*y*)

can return the values of variables ?*x*=92 and ?*y*=full matching their corresponding values in the belief memory.

6.2.5 Belief updaters

The belief memory includes beliefs that require abstract data for general decisions. This abstraction is done by the process called "belief updaters." Each belief updater is a process specialized in updating a category of beliefs and modifies the content of the belief memory using the services provided by the belief manager.

In Aerostack, there is a special belief updater called "common belief updater" maintaining an updated number of basic beliefs that are common for most robots, such as beliefs related to the position and battery charge.

Belief updaters periodically revise beliefs using dynamic information generated by execution processes (e.g., data from sensors); for example, the update frequency of these updates must be consistent with the time required for requesters to make decisions. The frequency of these updates is usually smaller than what the motion controllers used.

The proposed architecture has been conceived to include several belief updaters that are independent of each other, and thus a developer can flexibly add or remove specific belief updaters to a particular application according to operational needs.

7 Experimental evaluation

In this section, we summarize the evaluations of the method for the execution control system presented in this study. The evaluation includes two types of tasks: experimental flights and analysis of the integration effort. Experimental flights have been carried out to evaluate the performance of the method. The analysis of the integration effort has been done to verify if the method can be acceptable for potential developers who use the Aerostack software framework.

7.1 Experimental flights

7.1.1 Aerial mission

The execution control system was evaluated with the help of a type of aerial mission based on the competition rules of Autonomous Drone Racing (ADR) of IROS 2018 (International Conference of Intelligent Robots and Systems). The competition is a race with indoor autonomous flight challenges (e.g., frames to cross, and obstacles).

Fig. 8 shows an aerial robot and a set of frames in the experimental flights. We generated this scenario with the simulator RotorS (Furrer et al., 2016), which was used to perform a set of experiments helping refine the design of the initial versions of the execution control system.

Fig. 9 shows a real experimental flight corresponding to the same mission. Real experimental flights were conducted for the final version of the execution control system. In these experimental flights, we used an aerial vehicle, parrot bebop 2, and a laptop computer (the same computer used for simulated flights) equipped with CPU Intel i7-7700HQ, 8 cores, 2.8 GHz, and 16 GB RAM.

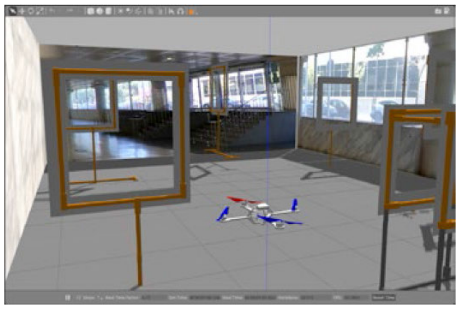

Fig. 8 Flight simulation using RotorS

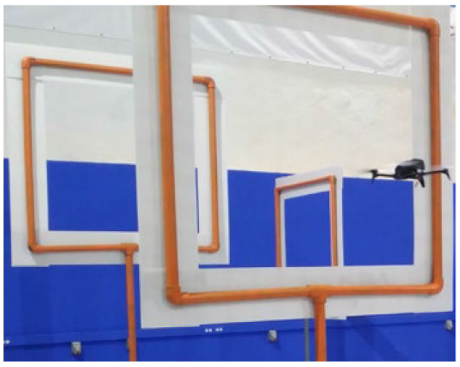

Fig. 9 A real flight experiment

A preliminary version of the execution control method presented in this study was validated with the second type of aerial mission related to surface inspection. It was useful in verifying the generality of the approach with another type of mission and in verifying the correct execution control in collaboration with an operator, where the operator provides assistance to the aerial robot to complete the mission. Details about this second type of experiment can be found in Molina et al. (2018).

7.1.2 Evaluation method

The method we used to evaluate the execution control system includes a first analysis of sequences of behavior activations and sequences of process executions corresponding to the experimental flights. The goal of this analysis was to verify that the system correctly translated the sequences of behavior activations into consistent changes in the configuration of running processes.

The performance of the execution control system was evaluated by measuring the time spent by the behavior coordinator in activating behaviors. This was measured as time τ_i, from the moment the coordinator receives a message to activate a behavior b_i to the moment the coordinator asks the execution controller of behavior b_i to start the execution. An aggregated value (e.g., average, minimum, or maximum) of the measurement set $\{\tau_1, \tau_2, \ldots, \tau_n\}$ (where n is the number of behavior activations) was used as a performance indicator for the execution control system.

The evaluation method verified that the trajectories generated by the aerial autonomous robot were correct according to the mission goal; i.e., they crossed all the frames in a reasonable period of time.

7.1.3 Evaluation results

The evaluation method was applied to six different real flights that performed the same type of mission. A dataset was generated with the following information: the temporal sequence of behavior activations including 126 behavior activations, the temporal sequence of process executions, and the 3D point coordinates of the aerial trajectories followed by the robot in different missions.

The sequences of behavior activation requests were automatically generated by a mission plan interpreter of Aerostack that used a mission plan specification (written in Python language) describing the tasks in the mission. This mission plan specified that the robot had to cross several frames whose location was only approximately known before the execution of the mission. Therefore, during the mission, the robot had to reach a certain point and search for a frame. When a frame was found, the robot approached the center of the frame to cross it and started this cycle again until all frames were crossed.

Fig. 10 shows a partial view (first 60 s) of the behavior activation sequence and running processes related to motion planning and motion control generated in a real experimental flight (experiment 6). Other behaviors and processes related to different functions, such as self-localization and mapping, are not presented in Fig. 10.

The sequence showed that when the behavior GO_TO_POINT was approximately activated at 1 s, two processes (motion controller and trajectory planner) were started by the execution control system

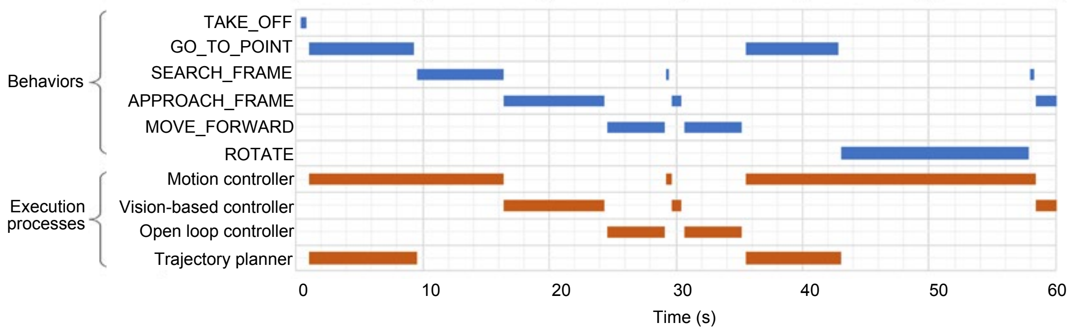

Fig. 10 Partial sequence of behavior activations (blue) and execution of processes (red)
References to color refer to the online version of this figure

to support the behavior execution. When the behavior GO_TO_POINT approximately terminated at 9 s, the execution control system correctly stopped the trajectory planner process and kept running the motion controller process. This is adequate for efficiency reasons, because the next behavior to activate SEARCH_FRAME used the motion controller process. The sequence showed that incompatible controllers (e.g., motion controller and vision-based controller) were correctly executed in alternation.

This type of sequence analysis was done for all the cases included in the evaluation dataset. The analysis demonstrated that all processes were consistently executed according to the mission goal.

Coordination time $\{\tau_1, \tau_2, \ldots, \tau_n\}$ was obtained from the evaluation dataset (n=126). The arithmetic mean coordination time was 119 ms and the minimum and maximum were 36 ms and 204 ms, respectively. These values showed that the execution control system which should be used by execution requesters worked at a frequency up to 1 Hz, which is adequate for making decisions during the development of a wide range of autonomous aerial missions in unknown environments (e.g., inventory missions and inspection missions).

Fig. 11 shows an example of a trajectory in experiment 6, and this trajectory was completed in 118 s. The autonomous robot correctly crossed all frames, and the trajectories generated in the rest of the real experimental flights also correctly crossed all frames.

Fig. 12 shows more details about the trajectory in experiment 6 to cross the first frame, and with different colors, the motion behaviors were active during each part of the trajectory (first 29 s).

For example, in the first part of the movement, the active behavior was TAKE_OFF. The next active behavior was GO_TO_POINT, which was activated to reach a high point from which the aerial robot can observe the frames well. Then the next active behavior was SEARCH_FRAME, which performed a prefixed movement until the image of a frame was detected with the help of the frontal camera. The aerial robot activated the behavior APPROACH_FRAME, which correctd the position of the drone; thus, it was in front of the frame and ready to cross. Finally, the robot activated the behavior MOVE_FORWARD to cross the frame.

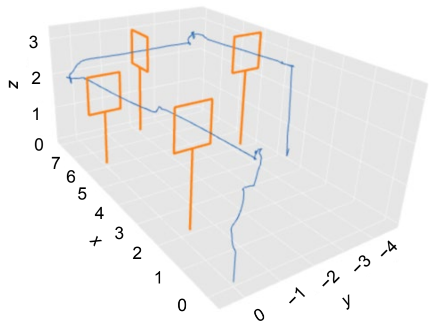

Fig. 11 An example of the trajectory generated in a real flight

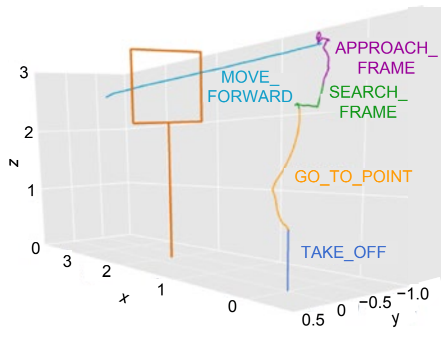

Fig. 12 Details of the trajectory to cross the first frame
References to color refer to the online version of this figure

7.2 Integration effort

To verify that the execution control method may be acceptable for potential developers who use the Aerostack software framework, we evaluated the integration effort of new behaviors using the following two measurements:

1. Programming effort. This is the amount of work in person-hours for software developers to integrate a new behavior.

2. Code size. This is the number of lines of code programmed to integrate a new behavior and its relation to the size of the reused code.

Table 2 shows the values associated with integrating five behaviors in Aerostack (the behavior SELF_LOCALIZE was implemented following two different approaches M1 and M2). To measure the programming effort, four tasks were considered: learning the behavior implementation (L), designing the execution controller (D), programming the execution controller (P), and validating the integration (V). These evaluation numbers corresponded to integration tasks performed by two different developers with more than six months in Aerostack.

Table 2 Evaluation numbers of programming effort in person-hours

Behavior	Evaluation number (person-hour)				
	L	D	P	V	Total
GO_TO_POINT	16	24	16	16	72
MOVE_FORWARD	0.5	1.0	5.0	3.0	9.5
SEARCH_FRAME	1	2	5	10	18
SELF_LOCALIZATION (M1)	1	1	8	4	14
SELF_LOCALIZATION (M2)	56	16	24	16	112

L: learning the behavior implementation; D: designing the execution controller; P: programming the execution controller; V: validating the integration

As expected, the effort significantly varied for different behaviors, due to their diverse complexity. The average value was 45.1 person-hours (5.6 person-days) and the maximum was 112 person-hours (14 person-days), which are acceptable according to the usability objectives of Aerostack.

The line number of the execution control system corresponding to the part which is common for all behaviors was 4754, including the following components: behavior coordinator, process manager, belief manager, common belief updater, and common classes. The code size to integrate a new behavior was estimated at 368 lines (the average value of execution controllers of the previous five behaviors), which is a significantly low value compared with the amount of code reused for the execution control system.

8 Conclusions

We have described the results of our work in the development of a general solution for execution control systems, which is one of the critical components of control architectures in robotics.

The solution shares parts of the design concepts of the current state of the art in robotic control architectures. We have conceived an original design to respond to the need to be parts of a software framework, practically useful and efficient for building real robotic systems.

The design combines two main ideas: (1) A distributed organization separates and encapsulates the execution details of each behavior helping developers add new behaviors with flexibility without affecting other behaviors; (2) There is a central coordination supported with the help of a catalog with compatibility conditions formulated at the behavior level (which is simpler than that at the process level). The catalog uses a declarative flexible representation to easily add information about new behaviors.

The experimental flights demonstrate that the method is valid for practical applications in aerial robotics and can be applied to specific systems with acceptable development efforts.

The solution is integrated in the Aerostack open-source framework (version 3.0). Consequently, its technical details are fully accessible to be consulted and reused in developing new applications in aerial robotics. Our design is supported by standards used in robotics (e.g., ROS), which facilitates its general use.

In spite of this, the method can be improved. The capabilities of the Aerostack executive system could be extended with a declarative language to formulate complex behaviors that use other behaviors. In addition, to execute more complex mission plans, the belief memory management could be extended with rich representations and good methods (e.g., representation with uncertainty, complex methods for persistency, and belief revision methods).

References

Alami R, Chatila R, Fleury S, et al., 1998. An architecture for autonomy. *Int J Robot Res*, 17(4):315-337. https://doi.org/10.1177/027836499801700402

Allgeuer P, Behnke S, 2013. Hierarchical and state-based architectures for robot behavior planning and control. Proc 8th Workshop on Humanoid Soccer Robots and 13th IEEE-RAS Int Conf on Humanoid Robots, p.1-6.

Arkin RC, 1998. Behavior-Based Robotics (Intelligent Robotics and Autonomous Agents). MIT Press, Cambridge, USA.

Bavle H, Sanchez-Lopez JL, de la Puente P, et al., 2018. Fast and robust flight altitude estimation of multirotor UAVs in dynamic unstructured environments using 3D point cloud sensors. *Aerospace*, 5(3):94. https://doi.org/10.3390/aerospace5030094

Bonasso RP, Firby RJ, Gat E, et al., 1997. Experiences with an architecture for intelligent, reactive agents. *J Exp Theor Artif Intell*, 9(2-3):237-256. https://doi.org/10.1080/095281397147103

Brooks R, 1986. A robust layered control system for a mobile robot. *IEEE J Rob Autom*, 2(1), 14-23.

Firby RJ, 1989. Adaptive Execution in Complex Dynamic Worlds. PhD Thesis, Yale University, New Haven, USA.

Furrer F, Burri M, Achtelik M, et al., 2016. RotorS—a modular gazebo MAV simulator framework. In: Koubaa A (Ed.), Robot Operating System (ROS). Springer, Cham, p.595-625. https://doi.org/10.1007/978-3-319-26054-9_23

Gat E, 1992. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. Proc 10th National Conf on Artificial Intelligence, p.809-815.

Gat E, 1996. ESL: a language for supporting robust plan execution in embedded autonomous agents. IEEE Aerospace Conf, p.319-324. https://doi.org/10.1109/AERO.1997.574422

Gat E, 1998. On three-layer architectures. In: Kortenkamp D, Bonnasso RP, Murphy R (Eds.), Artificial Intelligence and Mobile Robots. MIT Press, Cambridge, USA, p.195-210.

Ingrand F, Py F, 2002. An execution control system for autonomous robots. Proc IEEE Int Conf on Robotics and Automation, p.1333-1338. https://doi.org/10.1109/ROBOT.2002.1014728

Kortenkamp D, Simmons R, Brugali D, 2008. Robotic systems architectures and programming. In: Siciliano B, Khatib O (Eds.), Springer Handbook of Robotics. Springer Berlin Heidelberg, p.187-206. https://doi.org/10.1007/978-3-540-30301-5_9

Mittal S, Frayman F, 1989. Towards a generic model of configuration tasks. 11th Int Joint Conf on Artificial Intelligence, p.1395-1401.

Molina M, Suarez-Fernandez RA, Sampedro C, et al., 2017. TML: a language to specify aerial robotic missions for the framework Aerostack. *Int J Intell Comput Cybern*, 10(4):491-512. https://doi.org/10.1108/IJICC-03-2017-0025

Molina M, Frau P, Maravall D, 2018. A collaborative approach for surface inspection using aerial robots and computer vision. *Sensors*, 18(3):893. https://doi.org/10.3390/s18030893

Rodriguez-Ramos A, Sampedro C, Bavle H, et al., 2017. Towards fully autonomous landing on moving platforms for rotary unmanned aerial vehicles. Int Conf on Unmanned Aircraft Systems, p.170-178. https://doi.org/10.1109/ICUAS.2017.7991438

Rothenstein AL, 2002. A Mission Plan Specification Language for Behaviour-Based Robots. MS Thesis, University of Toronto, Toronto, Canada.

Rutten E, 2001. A framework for using discrete control synthesis in safe robotic programming and teleoperation. Proc IEEE Int Conf on Robotics and Automation, p.4104-4109. https://doi.org/10.1109/ROBOT.2001.933259

Sampedro C, Bavle H, Sanchez-Lopez JL, et al., 2016. A flexible and dynamic mission planning architecture for UAV swarm coordination. Int Conf on Unmanned Aircraft Systems, p.355-363. https://doi.org/10.1109/ICUAS.2016.7502669

Sampedro C, Rodriguez-Ramos A, Bavle H, et al., 2018. A fully-autonomous aerial robot for search and rescue applications in indoor environments using learning-based techniques. *J Intell Robot Syst*, p.1-27. https://doi.org/10.1007/s10846-018-0898-1

Sanchez-Lopez JL, Molina M, Bavle H, et al., 2017. A multi-layered component-based approach for the development of aerial robotic systems: the aerostack framework. *J Intell Robot Syst*, 88(2-4):683-709. https://doi.org/10.1007/s10846-017-0551-4

Suárez Fernández RA, Sanchez-Lopez JL, Sampedro C, et al., 2016. Natural user interfaces for human-drone multi-modal interaction. Int Conf on Unmanned Aircraft Systems, p.1013-1022. https://doi.org/10.1109/ICUAS.2016.7502665

Verma V, Estlin T, Jónsson A, et al., 2005. Plan execution interchange language (PLEXIL) for executable plans and command sequences. Int Symp on Artificial Intelligence, Robotics and Automation in Space, p.1-8.