TML: A LANGUAGE TO SPECIFY AERIAL ROBOTIC MISSIONS FOR THE FRAMEWORK AEROSTACK

MARTIN MOLINA

Department of Artificial Intelligence, Technical University of Madrid (UPM)
Campus de Montegancedo S/N 28660 Boadilla del Monte, Madrid, Spain

RAMÓN A. SUÁREZ- FERNÁNDEZ, CARLOS SAMPEDRO,
JOSE L. SANCHEZ-LOPEZ, PASCUAL CAMPOY

Centre for Automation and Robotics (CAR), UPM-CSIC
Calle Jose Gutierrez Abascal 2, 28024 Madrid, Spain

ABSTRACT

Purpose - The main purpose of this paper is to describe the specification language TML for adaptive mission plans that we designed and implemented for the open source framework Aerostack for aerial robotics.
Approach – The TML language combines a task-based hierarchical approach together with a more flexible representation, rule-based reactive planning, to facilitate adaptability. This approach includes additional notions that abstract programming details. We built an interpreter integrated in the software framework Aerostack. The interpreter was validated with flight experiments for multi-robot missions in dynamic environments.
Findings – The experiments proved that the TML language is easy to use and expressive enough to formulate adaptive missions in dynamic environments. The experiments also showed that the TML interpreter is efficient to execute multi-robot aerial missions and reusable for different platforms. The TML interpreter is able to verify the mission plan before its execution, which increases robustness and safety, avoiding the execution of certain plans that are not feasible.
Originality – One of the main contributions of this work is the availability of a reliable solution to specify aerial mission plans, integrated in an active open-source project with periodic releases. To the best knowledge of the authors, there are not solutions similar to this in other active open-source projects. As additional contributions, TML uses an original combination of representations for adaptive mission plans (i.e., task trees with original abstract notions and rule-based reactive planning) together with the demonstration of its adequacy for aerial robotics.

Keywords Mission plan specification; Autonomous aerial systems; Adaptive mission plans; Aerial robotic systems

Paper type Research paper

1. Introduction

To build aerial robotic systems with high levels of autonomy it is important to have tools to integrate multiple heterogeneous computational solutions (e.g., computer vision algorithms, actuator controllers, planning algorithms, etc.). This integration should be done in a way to be easily used by developers and operators of such a systems. This technical challenge has been one of our motivations for building the software framework Aerostack in our research group (`www.aerostack.org`) [Sanchez-Lopez et al., 2016; 2017]. This framework has demonstrated to be an effective tool for building different types of

aerial systems. Aerostack is currently an active open-source project with periodic software releases.

One of the important functions of this type of tool is to help developers specify a mission plan for aerial systems. In the initial version of Aerostack, this specification was based on writing programs and configuration text files. This method can be adequate for programmers who are familiar with Aerostack architecture, but it may not be appropriate for other kind of users. The main problem of this solution is that the developer must know many low-level technical details. The programs assume that the developer knows all these details and they are not adequately protected against errors. Therefore, this solution can be difficult to use and error-prone for other type of users.

To improve this, it is needed a solution to specify a mission easier to use and robust together with other practical requirements (e.g., efficiency, scalability, flexibility). In aerial robotics, there are applications that help operators to describe a mission with simple methods that are easy to use (e.g., waypoint lists). However, this type of method is insufficient to provide the necessary adaptability for autonomous flights in dynamic and complex environments. There are other methods with more flexible representations (e.g., task trees, finite state machines, rule bases, Petri nets, etc.) that have been used in other robotic systems, different from aerial systems. But, to the best knowledge of authors, there are not reliable tools available for the research community implementing these methods for aerial autonomous multi-robot systems.

As an answer to this need, this paper presents a mission specification language called TML for aerial robotics together with a reliable interpreter integrated in the Aerostack framework. TML combines a task-based hierarchical approach together with a more flexible representation (rule-based reactive planning) as a solution to formulate adaptive mission plans, with additional notions that abstract technical details at a programming level. We programmed an interpreter for this language with verification procedures to increase robustness. The interpreter was validated with real flight experiments in complex missions and it is freely available as part of the open-source framework Aerostack.

The remainder of the paper is structured as follows. First, the paper presents a discussion about the state of the art in mission plan specification in robotics. Then, the paper describes the characteristics of our proposed representation and the TML language. Then, the paper presents how the interpreter was implemented as a set of processes together with the framework Aerostack. Finally, we describe the results of the experimental evaluation based on real flights that demonstrate how TML can be used to formulate and execute adaptive mission plans for aerial robotics.

2. Related Work

A mission plan specification describes formally how to determine the order in which the robot must perform a sequence of simple actions to achieve a certain mission goal (e.g. a rescue mission). In aerial robotics, there are software applications that provide languages to the operators to specify mission plans. Many of these languages use lists of GPS waypoints with associated actions or commands. For instance, MP – Mission Planner[1] 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. However, the representations using waypoint lists are not able to adapt flexibly to mission circumstances [Santamaria et al., 2008; Schwatz et al 2014]. The specification is normally based on a fixed list of waypoints that cannot change dynamically in the presence of certain events.

To overcome this limitation, mission specification languages can follow other more flexible representations. For example, a popular approach in robotics is using finite state machines (FSMs). This representation has been used, for example, in languages such as the Behavior Language [Brooks 1990], or the Colbert language [Konolige, 1997]. Some more recent tools have also followed the FSM representation. For example, MissionLab[2] is an integrated tool that supports a graphical construction of state-transition diagrams together with a multiagent approach [MacKenzie 1997]. Currently Missionlab is an inactive project (the last release of the MissionLab was distributed in 2006).

The main problem of FSMs is that they are difficult to use in large complex models. They do not scale well and are difficult to maintain when the number of robot behaviors increases [Klöckner, 2013]. A particular state can be connected to any other state (anywhere in the model), which can be difficult to maintain when number of states is large. This has been sometimes called the state and transition explosion [Olsson 2016].

To solve this problem, other approaches have proposed modular and hierarchical representations such as hierarchical finite state machines (HFSM) [Yannakakis, 2000] [Kurt, Ozguner, 2013] or statecharts [Harel, 1987]. HFSM are modular and therefore more flexible to change and more comprehensible than FSMs. For example XABSL[3] (Extensible Agent Behavior Specification Language) is a programming language created in an open-source project [Loetzsch et al 2006] [Risler, 2009]. It was developed and demonstrated for soccer robots but it was conceived to be general for other types of robots (the last of release of XABSL was in 2009). Another proposal that uses HFSM is the State Control Library[4] and Behavior Control Framework[5] in NimboRo-OP [Allgeuer, Behnke, 2013]. This corresponds to open-source software validated in soccer robots, although there is not user documentation (the last release was in 2014).

Another hierarchical approach common in robotics is the task-based representation. The task is an intuitive common concept, easily understandable by general users, that has

[1] http://ardupilot.org/planner
[2] http://www.cc.gatech.edu/ai/robot-lab/research/MissionLab
[3] http://www.xabsl.de
[4] http://sourceforge.net/projects/statecontroller/
[5] http://sourceforge.net/projects/behaviourcontrol/

been already used in classical planning (e.g., HTN – Hierarchical Task Network). 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] (as a demonstrative prototype). Task-based specifications are intuitive descriptions so they are comparatively easier to specify and monitored than others [Fernandez- Perdomo 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: Halo 2 [Isla, 2005], Facade [Mateas, Stern, 2003], and Rockstar Games [Champandard, 2007]. 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]. [Olsson, 2016] shows the advantages of using behavior trees instead of FSMs with quantitative metrics.

In summary, the review of the research literature in robotics shows that there are different solutions to specify mission plans: waypoint lists, state-based specification and hierarchical approaches (task trees, hierarchical FSMs, behavior trees) besides others (workflow languages, imperative languages, Petri-nets, rule-based systems, etc.). Hierarchical representations are popular solutions for complex and adaptive missions. However, in the field of aerial robotics the available software tools are mainly based on waypoints (an insufficient representation for dynamic environments) and we have not found reliable tools for mission specification, for example in the form of active open-source projects, with a demonstrated adequacy for aerial autonomous multi-robot systems.

3. Representation of Adaptive Mission Plans

This section describes the representation that we propose to specify adaptive mission plans. Our representation is an integrated solution that combines a hierarchical task-based approach and rule-based reactive planning, together with concepts that abstract programming details.

It is important to note that we designed this representation to increase the degree of autonomy of the robot with respect to the operators. Operators should be able to ask robots to perform sets of tasks (formulated as a mission plan in a simple language easy to use) and the robot should be autonomous to respond to the request in two ways:

1. *Verification*. The robot should be autonomous to accept or reject the proposed tasks of the mission plan. For this purpose, the aerial robot should exhibit a certain degree of understanding of the requested tasks to verify their correctness and physical feasibility before the execution. The robot should also explain clearly to the operator the reasons that justify why certain requests are rejected.

2. *Adaptation*. The robot should be autonomous to adapt safely the execution of the plan to the specific details and unexpected events of dynamic environments. We use the adjective *adaptive* for mission plans to express that they should be general and flexible enough to be adapted to mission circumstances (for example, changes in the environment due to the presence of unknown obstacles, presence of other robots, etc.).

In addition, we designed the representation considering also additional practical requirements such as independence on aerial platforms, efficiency to be used in real flights of aerial robotics, and scalability to grow up easily with new functionalities.

3.1. *The task-based approach*

To describe this representation formally in we use the following notation. The hierarchical specification is as a tuple $H = \langle T, A, S, f, g, h \rangle$ where T is a set of tasks, A is a set of actions, S is a set of skills, and the functions $f : T \rightarrow P(T)$, $g : T \rightarrow A$, and $h : T \rightarrow P(S)$ represent respectively child nodes of a task (subtasks), action of a task and skills of a task. $P(T)$ is the power set of the set T.

3.1.1. *Task trees*

We specify a mission plan with a set of tasks organized in task trees. A task $t_i \in T$ specifies a piece of work to be done by the robot to achieve a desired goal in the mission. The name given to each task represents the activity to do to reach a goal such as *search a subject* or *enter the building* (i.e., the name of the task is not the goal expressed as a final state such as *the subject is found* or *the vehicle is inside the building*).

The task is used as a basic component to structure a mission with a modular organization. The mission is specified with a hierarchy of tasks and subtasks that describe the different parts of the mission to be done. An intermediate node t_i of the task tree with child nodes $T_i = \{t_1, t_2, \ldots, t_n\}$, $T_i \subset T$, is formalized with the function $f(t_i) = T_i$, where $t_i \notin T_i$. This represents that the goal of task t_i is achieved by doing tasks t_1, t_2, …, and t_n.

The task tree is a useful representation to establish the execution order of actions, for example, following a deep-first search strategy. However, to have a more flexible flow control in our approach, the nodes of the task tree accept control regime modifiers such as repetitions (i.e., the same task is performed several times) or conditional execution (i.e., a task is only performed if a condition is satisfied).

3.1.2. *Actions and skills*

In the task tree, a terminal node describes an action to do. We use the concept *action* $a_i \in A$ to express an elementary piece of work that the aerial robot is able to achieve directly

without the need of any further decomposition. Each action a_i is associated to a terminal task t_j, i.e., a task that is not decomposed into other subtasks. This can be formally expressed as $f(t_j) = \phi$ and $g(t_j) = a_i$. The following list is an illustrative set of motion actions: take off, go to a point, land, look at a point, flip, follow a trajectory, wait hovering a certain time, track an object and move in circles. Besides motion actions there are others such as: take a video, turn off the lights, memorize object image (to be tracked), memorize current point, say a sentence out loud, take a photo, send a message to other robots, etc. We assume that certain actions (e.g., actions that use the same actuators) are mutually exclusive, i.e., only one action can be performed at any given moment.

Actions can be specified with the help of specific parameters such as the following:

- Spatial references. For example, waypoints including spatial location (x, y, z) and optionally other values (e.g., speed, acceleration, and orientation).
- Temporal references with absolute time values (hour, minute, second) or relative values (before, after, overlap, etc.) expressing temporal constraints for certain tasks.

To complement actions, we use the concept of *skill* $s_i \in S$ that represents a particular robot's ability. The following list shows example skills: avoid obstacles, limit extreme movements, interpret ArUco visual markers, interpret voice sentences, and say out loud the current action.

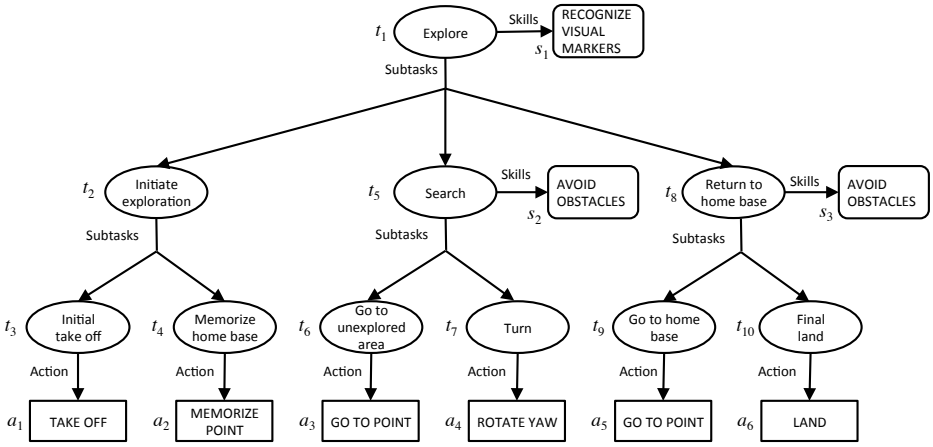


Figure 1: An example of task tree with actions and skills. Ellipses represent tasks, rectangles represent actions and rounded rectangles represent skills.

Skills can be active or inactive in a particular robot. The activation of a particular skill does not usually have a direct effect. Its effect is normally observed indirectly when an action is performed. Since skills have influence in the behavior of actions, we can understand skills as global modifiers for sets of actions.

The notion of skill is useful as an abstract concept to help operators express more easily what complex abilities should be active, without considering low-level technical details. Internally, a skill is automatically translated to a set of running processes. Thus, the activation of skills is associated to the increase of resource consumption (memory space, processing time, battery charge) so it is important to deactivate unnecessary skills when it is possible.

In the task tree, each task t_i has, optionally, a set of skills $S_i = \{s_1, s_2, \ldots, s_k\}$, $S_i \subseteq S$, formalized with the function $h(t_i) = S_i$, which means that the set of skills S_i must be active while the robot is doing task t_i. We also consider that skills could be activated with certain constraints such as: (1) distance, i.e., the skill is only activated when the distance between the position of the robot and a certain point is less than certain value, (2) delay, i.e., the skill is activated after a number of seconds, (3) yaw, i.e., the skill is only activated for a particular yaw.

3.2. *Reactive planning*

We combine the task-based solution with a rule-based reactive planning to provide more adaptability to changes of the environment. In general, reactive planning differs from classical planning in that it determines just one next action in every instant. This type of representation does not require having a representation of the effects of actions, as it is usually used in classical planning. This simplification is useful to cope with highly dynamic and unpredictable environments.

Reactive planning normally represents an internal state about the robot intentions (in our case, this is expressed with tasks). Actions are selected based on the internal state (the current task) and conditions about the external world state [Downs, Reichgelt, 1991]. Thus, for a given world state, different actions can be selected depending on the internal state. Both, the internal state and the conditions about the world, are normally represented using high level qualitative representations that abstract details from sensor data.

To represent conditions about the world state, we use the concept of event. An *event* is the occurrence of a significant change in the state of the external environment or the state of the own robot. Events can be related to normal situations (e.g., the detection of a specific visual marker) or undesired situations (discharged battery, etc.).

In our representation, the mission specification includes event handlers that define what to do in the presence of some events. Event handlers are formulated as condition-action rules, a typical representation used in reactive planners:

$$p_1 \wedge p_2 \wedge \ldots \wedge p_n \rightarrow q_1 \wedge q_2 \wedge \ldots \wedge q_m$$

The condition part of the rules $\{p_i\}$ includes conditions about the presence of events and/or conditions about what is the current task. The action part of the rules $\{q_j\}$ includes requests about: additional skills to be active, additional actions to be done, and/or change the normal execution flow (abort mission, abort task, jump to task, etc.).

3.2.1. *Multi-robot communication based on events*

We use events with reactive planning as a solution for multi-robot communication. A message sent from robot *A* to robot *B* is understood by robot *B* as a particular type of event (e.g., a public event). The mission plan of robot *B* captures the event with the corresponding event-handler that describes how robot *B* must react to its presence of such an event. To send a message, we follow two alternative solutions:

- There are specific actions to communicate with other robots. For example, the action *send message* sends a text message to other robots. This action includes different arguments such as the destination robots, the message, etc.

- We can assume that certain prefixed events detected by one robot (e.g., recognize the presence of a particular visual marker) are always automatically communicated to other robots.

Section 5 describes how we implemented this solution (with the components called event detector and event publisher). Section 6 presents an experimental example that illustrates how two robots communicate using this approach.

4. The TML Language

Based on the described approach for mission specification, we designed the language called TML (Task-based Mission specification Language). TML uses XML syntax to be readable by both humans and machines. Figure 2 shows a complete example of a mission specification using TML language, corresponding to the example presented in figure 1.

According to the TML syntax, a mission is defined with the tag <*mission*> and the attribute *name*. The body of the mission is separated in two parts: tasks and event handlers. The first part includes a task tree and each task is specified using the tag <*task*> and the attribute *name*.

The body of a terminal task specifies an action (e.g., `TAKE_OFF`, `LAND`, etc.) with the tag `<action>` and the attribute `name`. An action can include optionally arguments. Arguments use the tag `<argument>` and the attributes {`name`, `value`}. Table 1 shows a partial list of actions used in TML language. The body of a terminal task can specify several skills to be active with the tag `<skill>` and the attribute `name`. The available actions and skills used by TML can be increased easily with new actions and skills elements without affecting the central structure of representation of TML language.

Table 1: Example actions used by TML.

Action	Description
FLIP	The vehicle performs a flip in a certain direction (argument "direction" with the values {front, back, right, left}). The direction by default is to the front.
GO_TO_POINT	The vehicle moves to a given point. The point can be expressed using absolute coordinates (argument "point") or coordinates relative to the vehicle (argument "relative point").
LAND	The vehicle descends vertically (through the z axis) until it touches the ground. It is assumed that the ground is static.
ROTATE_YAW	The vehicle rotates the yaw a number of degrees (argument "angle").
STABILIZE	The vehicle tries to cancel all the perturbations and turbulences that may affect its system such as movement speeds and attitude speeds.
TAKE_OFF	The vehicle takes off from its current location to the default altitude. If the vehicle is flying, this action is ignored.
WAIT	The vehicle waits on the air for a specified number of seconds (argument "time").

```
<mission name="Exploration mission">
  <task name="Explore">
      <skill name="RECOGNIZE_VISUAL_MARKERS"/>
      <task name="Initiate exploration">
          <task name="Initial take off">
              <action name="TAKE_OFF"/>
          </task>
          <task name="Memorize home base">
              <action name="MEMORIZE_POINT"/>
                 <argument name="coordinates" label="HOME"/>
              </action>
          </task>
      </task>
      <task name="Search">
          <skill name="AVOID_OBSTACLES"/>
          <task name="Go to unexplored area">
              <action name="GO_TO_POINT">
                <argument name="coordinates" value="(6.0, 4.0, 1.0)"/>
              </action>
          </task>
          <task name="Turn">
              <action name="ROTATE_YAW">
                <argument name="angle" value="180"/>
              </action>
          </task>
      </task>
      <task name="Return to home base">
          <skill name="AVOID_OBSTACLES"/>
          <task name="Go to home base">
              <action name="GO_TO_POINT">
                <argument name="coordinates" label="HOME"/>
              </action>
          </task>
          <task name="Final land">
              <action name="LAND" />
          </task>
      </task>
  </task>
  <event_handling>
      <event name="Land command recognized">
          <condition parameter="visualMarker" comparison="equal" value="3"/>
          <action name="LAND"/>
          <termination mode="ABORT_MISSION"/>
      </event>
  </event_handling>
</mission>
```

Figure 2: Simple example of mission specification in TML language.

The body of a non-terminal task can include skills that are active during the task execution. The body of a non-terminal task does not include actions. Instead, the body includes one or several simpler tasks. The linear sequence of task execution can be modified with repetitions and conditions. For example, the tag `<repeat>` and the attribute `times` is used to repeat a task a number of times. The tag `<condition>` and the attributes {`parameter`, `comparison`, `value`} are used to establish a condition about the presence of an event to execute a task. The allowable values for the attribute comparison are {`equal`, `less than`, `less than or equal to`, `greater than`, `greater than or equal to`, `not equal to`}. For example:

```
<task name="Flip three times if green is observed">
  <condition parameter="observed color"comparison="equal" value= "green"/>
  <task name="Flip three times" times="3">
     <action name="FLIP"/>
  </task>
</task>
```

The second part of a mission is a list of event handlers. This part of the specification is defined with the tag `<event_handling>`. Each event handler describes how to react to a particular event. Events are defined with the tag `<event>` and the attribute `name`. Each event includes a list of conditions (in conjunctive form), a list of actions to be done and an ending step defined with the tag `<termination>` and the attribute `mode`. Events can use a particular type of condition with the attribute `currentTask`. This is useful to express that the event is considered while a particular task is being executing.

5. Execution of TML Plans with Aerostack

This section describes the solution that we designed and implemented for the software framework Aerostack [Sanchez-Lopez et al., 2016; 2017] to execute mission plans formulated in TML language.

We designed an architecture of processes divided into three parts: the TML interpreter, a set of executive processes and the Aerostack interface. This architecture is supported at the implementation level by the middleware ROS (Robot Operating System). Figure 3 shows our architecture as a block diagram. Each rectangle represents a process (mission plan interpreter, action interpreter, etc.) implemented as a ROS node. Each process has a set of ports (with arrows) that describes how its inputs and outputs are connected to other processes. In the figure, we use two different notations for the ports: squared ports that use a publish/subscribe communication model and rounded ports that use a request/reply communication model (see ROS[6] for a detailed description about this type of inter-process communication). Next sections describe all these processes in more detail.

[6] http://www.ros.org

5.1. *The TML interpreter*

The TML interpreter is implemented with a process called *mission plan interpreter*. This process reads a text specification in TML language written by the human operator and verifies the correct description of the mission plan. Then, during the plan execution, the interpreter adapts safely the execution of the plan to mission circumstances, generating step by step the next action to perform with the set of skills to be active.

5.1.1. *Interpretation algorithm*

During the plan execution, the mission plan interpreter uses an algorithm that follows two basic strategies:

- A task-driven strategy, i.e., the execution follows the sequence of tasks established by the task tree and translates them into specific action requests to be done and skills to be active, and
- An event-driven strategy, i.e., when the interpreter is waiting until a requested action is completed, the interpreter analyzes the presence of specific events to react according to the event handlers.

Algorithm 1 shows how these two strategies are implemented. The task-driven strategy is described as a loop (line 2) that covers all tasks according to a sequence established by line 12 (next task). This line can be implemented as a function call that generates step by step the sequence of tasks following a depth-first control strategy, although this can be modified with repetitions and conditions defined for some tasks.

Algorithm 1. Interpretation of a mission plan specification

1. $t_i \leftarrow$ root node t_1 of the task tree
2. **while** (t_i not empty) **do**
3. **if** $f(t_i) = \phi$ (t_i is a terminal node) **then**
4. $T_{1i} = \{t_k / t_k$ belongs to the path that goes from the root t_1 to the node $t_i\}$
5. $S_{1i} = \{s_j / t_k \in T_{1i}\ ,\ h(t_k) = S_k\ ,\ s_j \in S_k\}$ (skills s_j of all tasks t_k in T_{1i})
6. request the activation of the set of skills S_{1i}
7. request the execution of action $a_i = g(t_i)$
8. **while** (action a_i is running) **do**
9. check conditions of event handlers
10. **if** (event handler e satisfies its condition) **then**
11. interpret the requests of handler e
12. $t_i \leftarrow$ next task of t_i

The event-driven strategy is described in line 9 where event handlers are checked. We assume that when more than one handler condition holds in a given instant, the conflict is solved selecting the first handler according to the order they present in the specification. This gives to the operator the possibility to establish priority orders. Line 11 corresponds to the interpretation of the requests of the handler. As a consequence of this interpretation it is possible to: request additional skills to be active, request additional actions to be done, and/or change the execution flow (abort mission, abort task, jump to task, etc.).

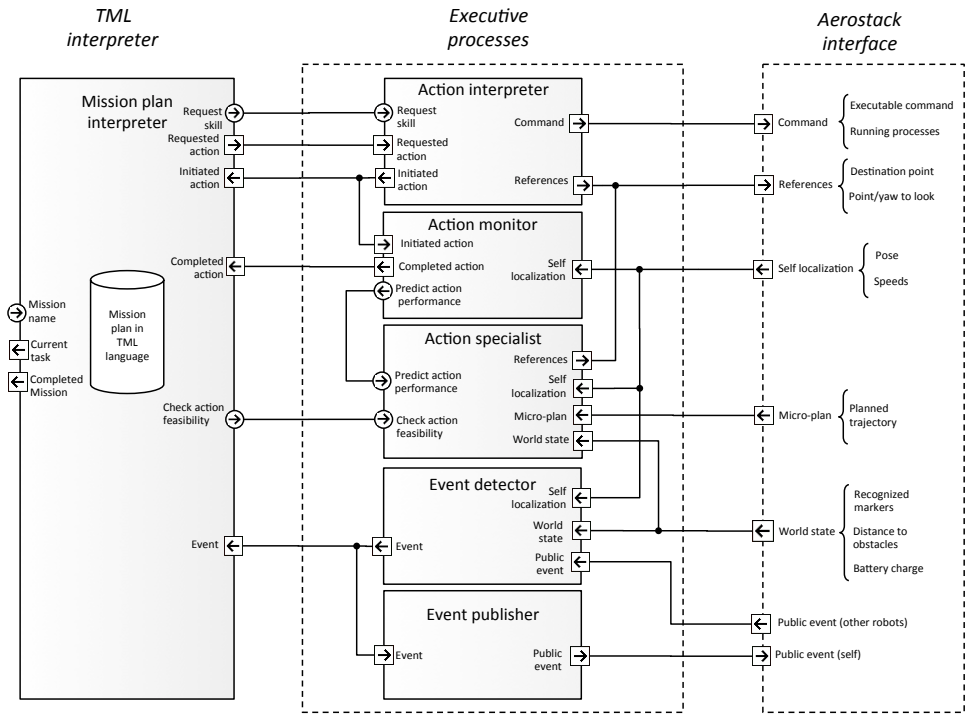


Figure 3: Block diagram of the processes used to execute mission plans formulated in TML language.

5.1.2. *Verification of mission plans*

The interpreter of TML language verifies the mission plan before the plan is executed. This is important to increase the degree of robustness and safety, avoiding the execution of incorrect plans. For example, TML performs the validation of the language, which corresponds to the lexical, syntax and semantic validation of the mission specification according to the grammar defined for the TML language.

In addition to language validation, it is important to consider also the verification of the physical feasibility, which checks if the specified mission can be performed in practice considering constraints of the physical world (this validates, for example, that a point to reach is not too close to an obstacle or that the distance to cover is not too long for the capacity of the battery). To carry out the verification of physical feasibility it is necessary to use models that represent knowledge of physical laws and the environment. In our approach, we separate the verification of physical feasibility into (1) the verification of an

individual action, and (2) the verification of the complete mission taking into account the temporal evolution of the whole set of actions and skills.

Section 5.2.3 describes how we verify an individual action using a constraint-based approach. The verification of the complete mission, taking into account the temporal evolution, is a procedure that requires more complex models [Rothwell, et al., 2013; Humphrey, et al., 2014]. Figure 4 shows an example of an experimental representation that we used to explore how to verify the temporal verification of plans. The example shows a logic-based representation using event calculus. Here, a TML plan is automatically translated into a logic representation. This representation also includes logic axioms about physical reasoning. Then, a reasoning tool uses logic inference to evaluate if the plan is feasible.

$$
\begin{aligned}
&\mathit{HoldsAt}\ (\mathit{InAir}\ (\mathit{robot}),\ t) \wedge \mathit{HoldsAt}\ (\mathit{StandingOn}\ (\mathit{robot},\mathit{xyz1}),\ t) \wedge \\
&\mathit{HoldsAt}(\mathit{MaxVelocity}\ (\mathit{robot},\mathit{velocitymax}),\ t) \wedge \\
&\mathit{HoldsAt}(\mathit{DistanceOneToTwo}(\mathit{xyz1},\mathit{xyz},\mathit{distance}),\ t) \wedge \\
&\mathit{HoldsAt}(\mathit{MoveOk}(\mathit{robot},\mathit{distance}),\ t) \wedge \mathit{HoldsAt}(\mathit{ObstacleOk}(\mathit{robot},\mathit{xyz1},\mathit{xyz}),t) \wedge \\
&\mathit{Happens}(\mathit{MoveTo}(\mathit{robot},\mathit{xyz},\mathit{velocity}),t) \wedge \mathit{velocity} \leq \mathit{velocitymax} \\
&\rightarrow \mathit{Terminates}(\mathit{MoveTo}(\mathit{robot},\mathit{xyz},\mathit{velocity}),\ \mathit{Stabilize}(\mathit{robot}),\ t)
\end{aligned}
$$

Figure 4. Example of logic based representation using event calculus.

To analyze the adequacy of this approach in aerial robotics, we used DEC reasoner, a software tool for discrete event calculus [Mueller, 2004; Mueller, 2014]. A problem of this solution is that it needs important computational resources, which could be acceptable only in some specific cases, before the actual execution of the mission plan.

An alternative to the previous solution for temporal verification is the use of physical simulators. In this case, a simulator executes virtually the mission plan (under certain environmental assumptions) and an evaluator process verifies if the mission is carried out correctly. The environmental assumptions can be divided in several possible scenarios and the simulation can be repeated for all of them. We believe that this simulation-based alternative is a possible solution to be used in the future, due to the availability of reusable specialized software components in robotics for simulation and the increasing computational power.

5.2. *Executive processes*

To coordinate the correct execution of the actions requested from the mission plan interpreter, we designed a set of executive processes. These processes create a separation between two representation levels:

- The symbolic level, where the mission goals are described with intuitive and more abstract symbols (actions, skills and events), and

- The execution level, where the goals are described with concepts that are near the execution, using technical details (e.g., executable commands, running processes, quantitative references, etc.).

This separation is important to have a more intuitive language for the operator that abstracts programming details and simplifies the way the mission is described in TML language. The following sections describe the executive processes in more detail.

5.2.1. *The action interpreter*

The action interpreter accepts requests to execute actions and translates them into descriptions at the execution level. In particular, the action interpreter performs the following main functions:

- Translate requested actions (e.g., take off, move to a point, etc.) into specific execution level commands and quantitative references.
- Translate desired active skills (e.g., active the interpretation of Aruco visual markers) into running processes.
- Guarantee the consistency of requested actions and skills.

The action interpreter verifies if a request to active a skill is compatible with previous skill requests. If it is compatible, the action interpreter approves the request and memorizes it (the skill is not activated yet). Otherwise, the action interpreter rejects the request indicating incompatible previous requests. The interpreter of mission plans can request additional skills, one by one. Then, the interpreter of mission plans can request an action to be performed that is translated by the action interpreter into quantitative references and a command that includes the executable command to be done with the processes that need to be running.

5.2.2. *The action monitor*

The action monitor supervises the correct execution of initiated actions and detects if they have finished successfully or they have failed. For example, if the requested action is to go to a certain point, the action monitor verifies periodically the distance between the robot and the destination point and, when the distance is less than a threshold (established by a configuration parameter), the action monitor notifies that the behavior has been completed.

5.2.3. *The action specialist*

The action specialist verifies the physical feasibility of actions. This is useful to anticipate if a tentative action is feasible, according to the current situation. For example, the action specialist can verify in advance that a certain spatial point is too far to be reached, considering the current charge of battery.

The action specialist verifies the physical feasibility of individual actions using a constraint-based representation. This approach uses the following elements:

- *Variables* $\{x_i\}$ represent the dynamic values of physical references and magnitudes (e.g., destination point, current charge of battery, etc.).
- *Parameters* $\{k_i\}$ represent constant values for physical magnitudes related to the performance of the robot such as maximum speed, battery consumption rate of the vehicle, etc. Parameters can be divided into vehicle-independent parameters (general for any kind of vehicle, based on common sense knowledge about aerial robotics) or vehicle-specific for each type of vehicle.
- *Functions* $\{f_i\}$ represent spatio-temporal and motion functions (see Table 2) such as the length of a trajectory, distance to the closest obstacle, maximum distance covered with certain battery charge, required speed to reach a point at certain time, etc.
- *Constraints* $\{c_i\}$ are conditions about the robot and the physical world that must be satisfied.

Table 2. Example functions used in the verification model.

Function	Description
Distance(x, y)	Distance from point *x* to point *y*
DistanceBattery(x, y)	Maximum distance covered with battery charge *x* and consumption rate *y*
DistanceObstacle(x)	Distance from point *x* to the closest obstacle
Length(x)	Length of trajectory *x*
Speed(x, y, z)	Required speed to departure from point *x* and arrive at point *y* at time *z*
Trajectory(x, y)	Trajectory from point *x* to point *y* (generated by a trajectory planner)

For example, consider the following three conditions: the destination point must be safe from obstacles, there must be enough battery for the movement, and the destination point must be reachable at an acceptable speed. This can be represented with the following three constraints:

$$c_1: DistanceObstacle(x_2) > k_2$$
$$c_2: Length(Trajectory(x_1, x_2)) < DistanceBattery(x_3, k_1)$$
$$c_3: Speed(x_1, x_2, x_4) < k_3$$

where the variable x_1 is the current point, x_2 is the destination point, x_3 is the current battery charge, and x_4 is the planned time of arrival to the destination; and the parameter k_1 is the battery consumption rate, k_2 is the minimum free acceptable space between obstacle and vehicle, and k_3 is the maximum speed of the vehicle.

The proposed representation can be used to verify the feasibility of individual actions in the following way. For a given action, the verification procedure reviews the set of constraints that correspond to categories to which the given action belongs. For example, there is a set of constraints for actions related to rotation motions, another set of

constraints for actions related to translation motions, etc. This type of model is generic to be reusable for different physical platforms and scalable to include more constraints in the future. The robot-specific parameter values must be manually calibrated (or automatically obtained with machine learning methods) for each type of vehicle.

The action specialist is also able to predict physical magnitudes of certain actions such as expected required time, expected distance to cover, amount of battery to consume, required free space, etc. It is important to know, that this estimation is approximate, i.e. it is done using inexact models and help to find more efficiently the solution, anticipating certain clear solutions.

5.2.4. *The event detector*

The event detector identifies the presence of significant events to be used by the mission plan interpreter. For example, the event detector reads the outputs of perception algorithms (e.g., recognition of visual markers) and generates symbolic descriptions of events. The TML interpreter uses these descriptions and reacts to them according to the event handlers of the mission.

5.2.4. *The event publisher*

The event publisher implements a multi-robot communication method based on events. The objective of the event publisher is to determine if an event that has been recognized by a robot must be sent to other robots. Algorithm 2 describes in detail how this communication process works with the event publisher in combination with the event detector.

Algorithm 2. Multi-robot communication based on events

1. let $R = \{R_1, \ldots, R_n\}$ the set of robots of the system
2. **for each** robot $R_i \in S$ **do**
3. robot R_i analyzes the environment with the event detector D_i
4. **if** (event detector D_i identifies a local event e_i) **then**
5. robot R_i analyzes event e_i with the event publisher P_i
6. **if** (event publisher P_i decides to publish e_i for robots $S = \{R_k\}$, $k \neq i$) **then**
7. publish event e_i for robots $S = \{R_k\}$
8. **for each** event e_j published by another robot $R_j, j \neq i$ **do**
9. robot R_i analyzes event e_j with the event detector D_i
10. **if** (event detector D_i decides that e_j is relevant for robot R_i) **then**
11. D_i creates a new local event e_j' for R_i

According to this algorithm, in a particular situation, a robot R_i may analyze the environment using the event detector D_i to identify a local event e_i. Then, the event publisher P_i of robot R_i may determine that this detected event must be sent to a set of robots $S = \{R_k\}$, $k \neq i$. This decision is based on the category of the event e_i and the social model of robot R_i. On the other hand, if robot R_i receives a public event e_j detected

and published by robot R_j, the detector D_i of robot R_i analyzes e_j to determine whether this event is relevant for robot R_i or not. If e_j is relevant, the event detector D_i creates a new local symbolic event e_j' for robot R_i to be processed by the interpreter to react according to the event handlers of the mission specification. The experiment described at the end of this paper shows an example of this multi-robot interaction where the event is related to the recognition of a searched subject.

5.3. *Aerostack interface*

Aerostack [Sanchez-Lopez et al., 2016; 2017] supports the detailed execution of TML plans and the operation with aerial platforms. The executive processes of the TML interpreter interact with Aerostack to use functionalities related to perception, commands at the execution level, specialized planning tasks (e.g., trajectory planning) and communication with the operator and with other robots. The interaction with Aerostack is done based on the inter-process mechanisms provided by ROS (Robot Operating System).

We integrated the TML interpreter and the executive processes in Aerostack as additional open-source components of the Aerostack library. The integration with Aerostack was important to satisfy certain practical requirements. For example, since Aerostack is a software framework platform independent, the TML implementation also has this property. Only certain platform-dependent parameters must be calibrated for each type of vehicle. Aerostack also provides flight proven motion controllers and computer vision algorithms that provide the required efficiency for the TML execution to be used in real flights.

6. Experimental Evaluation

TML language has been tested in real flights using different mission cases with various degrees of complexity and with several interacting robots. Some examples are the following:

- We used TML to represent complex missions corresponding to recent competitions in aerial robotics, such as the indoor competition of IMAV 2016 [Molina et al., 2016]. These experiments proved that TML has an adequate level of expressivity to formulate adaptive mission plans to operate in dynamic environments. In addition, TML showed scalability to accept new actions and events for the different missions, keeping the basic design of the interpreter.
- We used TML as a language for our students in our university to learn practical aerial robotics. This experience showed that TML was an easy language to learn and use. Besides the members of our own research group, other users of Aerostack have operated with aerial robots with TML in their own projects and they have reported the correct operation of the language to formulate mission plans.

Figure 5: Demonstrative mission where two drones search in an autonomous flight for a hidden subject in a spatial area with unknown obstacles. Visual markers (ArUco) are used by the drones for self-positioning, obstacle detection and subject detection.

This section describes an experiment based on a representative example of a mission execution to demonstrate and illustrate the capabilities of TML. The experiment described here is based on a search and rescue mission.

6.1. *Mission Set-up*

In this mission, several autonomous drones depart from a rescue equipment base to search for a subject. The rescue equipment operators have previously defined the regions where they wanted the drones to search for the subject, so each drone covers a different search area. The drones autonomously navigate to these areas avoiding collisions with obstacles and with other drones in narrow areas. Once a drone detects the subject, it lands to stay with the subject and the other drones return to rescue equipment base.

To carry out the experiment, we used two physical platforms AR Drone 2.0. More drones and other platforms can be used in this mission, but since they have the same TML representation, two drones is enough to illustrate the TML operation in a multi-robot system and this simplifies the practical execution of the experiment. We also used two Unix based laptops, with Wi-Fi and Ethernet connection. The two laptops were connected in a LAN using the Ethernet interface and a switch. Every aerial platform was individually connected to an associated laptop by means of a Wi-Fi connection.

We defined the mission with robots that use visual markers (ArUco markers) to detect obstacles and the searched subject. We used these markers also for simplicity although

other methods for localization and mapping could be also used. In more detail, the mission is as follows:

- Two drones take off at the same time from one side of the spatial area.
- Each drone covers a different search area defined with an origin point and a destination point.
- Unknown obstacles are present and each drone must detect them and avoid them. In addition, each drone must avoid collisions with the other drone. The drones must cross narrow areas and they must decide how to enter in the appropriate order to avoid collisions between them.
- When a drone recognizes the presence of the subject, it sends a message to the other drone, which returns home and land.
- The drone that finds the subject lands to stay near the subject.

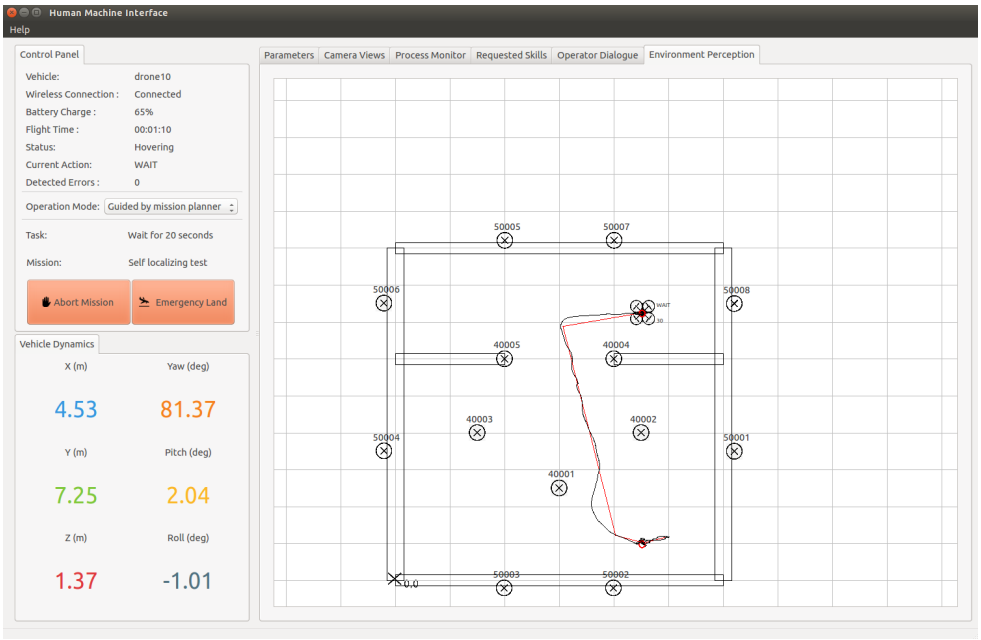


Figure 6: Example of the Aerostack user interface showing dynamically the movement of one of the two drones. The red line shows a planned trajectory. The black line corresponds to the actual flight.

6.2. *Results*

As a first result, the experiment shows that TML language was expressive to represent this aerial mission. Appendix A shows the complete specification in TML language for this mission. During the preparation of the mission, the experiment also demonstrated that the interpreter was able to verify correctly the language to identify wrong descriptions and explain adequately the detected errors to the operator.

6.2.1. *Multi-robot operation*

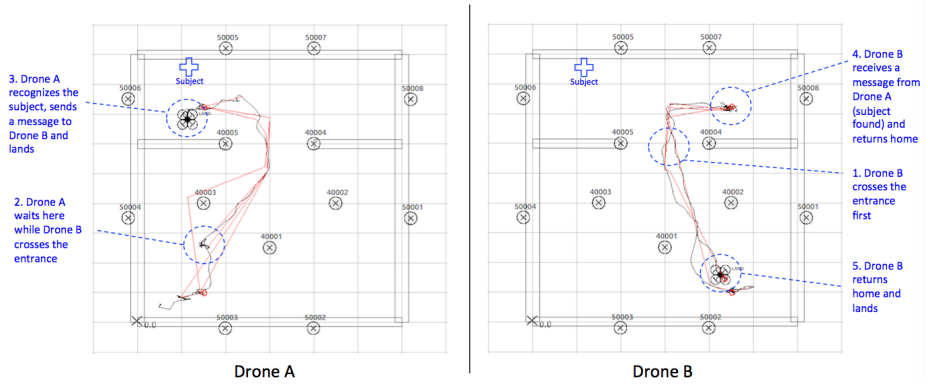


Figure 7: Complete trajectories carried out by both drones. They are performed during the same mission but, for the sake of clarity, we present them here separately here in two parts.

Figure 7 shows the trajectories followed by the two drones. Drone A and drone B move from their starting point to the destination point. Drone B crosses the entrance before drone A (step 1 in the figure). In its way, drone A has to wait until drone B crosses the narrow entrance (step 2 in the figure). The figure shows several red lines, corresponding to the tentative trajectories generated by drone A before crossing. Then, drone A recognizes the subject (step 3). This event is sent to drone B that starts returning home (step 4). Finally, Drone B arrives home and lands (step 5). Note that the global behavior of this multi-drone system is an emergent behavior that is not explicitly programmed, but it is a consequence of the interaction between drones and with the dynamic environment.

The experiment illustrates how the interaction between several drones can be managed through the use of event handlers in TML. In this experiment, the TML mission includes the following event handler:

```
<event name="Subject recognized by other drone">
  <condition parameter= "RECOGNIZED_ARUCO_MARKERS_BY_OTHERS"
             comparison="includes" value="2" />
  <termination mode= "NEXT_TASK" />
</event>
```

Here, the parameter `RECOGNIZED_ARUCO_MARKERS_BY_OTHERS` is a parameter whose value is generated by the event detector by reading messages received from other drones.

6.2.2. *Performance of the execution*

In the example presented here, our system was running on two computers: (1) computer A with an Intel I7-6560U (2.20 GHz, 4 cores) and 16 GB of memory and (2) computer B with Intel i7-4700HQ (2.4 GHz, 4 cores) and 4 GB of memory. Aerostack uses asynchronous multitasking, where different processes run concurrently, with inter-process communication provided by ROS. In this example, one single aerial robot was

operated with 31 processes executed simultaneously and the ROS messages were published on 67 different ROS topics.

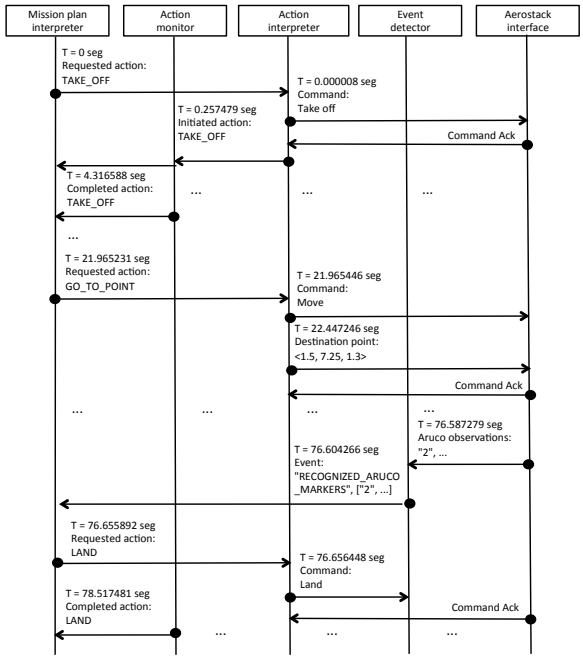


Figure 8: Example of inter-process communication developed in the experiment.

Figure 8 shows part of the inter-process communication between some processes in the experiment. This example shows only a few messages for illustrative purposes (the complete experiment includes a number near 1000 messages). The example shows, for example, how the mission plan interpreter sends messages to the action interpreter. The example also illustrates how the event corresponding to the subject detection triggers the landing action by the interpreter. In the figure, the time stamps indicate (in seconds) the delay of the sequence of messages. Even with this amount of processes and information exchanged, the solution worked fluidly and efficiently in real time.

7. Conclusions

In this paper, we have presented TML, a computer language that we designed to specify mission plans for aerial robots in the software framework Aerostack. TML shares some representation characteristics with other languages for mission plan specification (e.g., task trees and actions) but it also uses an original combination of representations easy to use and more adaptive to changes in dynamic environments (e.g., skills, and rule-based reactive planning).

We reviewed other proposals for mission plan specification in robotics. However, we found that they were insufficient for our needs in Aerostack because either (1) they use representations that are not adequate for dynamic environments (e.g., waypoint lists), or (2) they are theoretical approaches or partial prototypes that have not fully demonstrated their practical utility in aerial robotics. In contrast, TML has shown its applicability and practical utility in real flights of aerial multi-robot missions in dynamic environments.

Another important contribution of TML language is that it is part of an active open-source project. In consequence, its interpreter is freely available to be used by the community of developers in robotics. As a result of the literature review in this field, we have not found any other active open-source project with a similar capability to TML for dynamic environments to be used in aerial robotics.

Since TML is part of the Aerostack, we plan to generate periodically new releases of this specification language with improvements and extensions. For example, we plan to extend TML with additional actions and skills for aerial robotics. Based on the intuitive hierarchical structure of mission plans provided by TML, we also plan to design and build a graphical user interface to facilitate the creation of mission plans by manipulating graphical objects.

Acknowledgments

This research work 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 authors would like to thank the members of our research group CVAR (Computer Vision and Aerial Robotics) for their help in software programming and evaluation with real flights: David Palacios, Adrian Diaz-Moreno, Guillermo de Fermin, Alberto Camporredondo and Carlos Valencia.

Appendix A. Mission Specification in TML Language

This appendix shows the complete mission plan specification in TML language that we used for the experiment presented in this paper.

```
<mission name="Search and rescue">

  <task name="Prepare departure">
    <skill name="RECOGNIZE_ARUCO_MARKERS"/>
    <task name="Initial take off">
      <action name="TAKE_OFF" />
    </task>
    <task name="Wait one second">
      <action name="WAIT">
        <argument name="duration" value="1"/>
      </action>
    </task>
    <task name= "Adjust position at the initial point">
      <action name="GO_TO_POINT">
        <argument name="coordinates" value="(5.0, 7.5, 1.3)"/>
      </action>
```

```
    </task>
    <task name="Stabilize position">
      <action name="STABILIZE"/>
    </task>
    <task name="Memorize home base">
      <action name="MEMORIZE_POINT">
        <argument name="coordinates" label="HOME" />
      </action>
    </task>
  </task>

  <task name="Search subject">
    <skill name="RECOGNIZE_ARUCO_MARKERS"/>
    <skill name="AVOID_OBSTACLES"/>
    <task name= "Go to point (5.0,1.5)">
      <action name="GO_TO_POINT">
        <argument name="coordinates" value="(5.0, 1.5, 1.3)"/>
      </action>
    </task>
    <task name="Wait for 30 seconds">
      <action name="WAIT">
        <argument name="duration" value="30"/>
      </action>
    </task>
  </task>

  <task name="Complete search">
    <skill name="RECOGNIZE_ARUCO_MARKERS"/>
    <skill name="AVOID_OBSTACLES"/>
    <task name="Return to home base">
      <task name="Stabilize before turning">
        <action name="STABILIZE"/>
      </task>
      <task name="Turn 90 degrees">
        <action name="ROTATE_YAW">
          <argument name="orientation angle" value="90"/>
        </action>
      </task>
      <task name="Stabilize before returning">
        <action name="STABILIZE"/>
      </task>
      <task name="Go to home base">
        <action name="GO_TO_POINT">
          <argument name="coordinates" label="HOME" />
        </action>
      </task>
      <task name="Final land">
        <action name="LAND"/>
      </task>
    </task>
  </task>

  <event_handling>

    <event name="Subject recognized">
      <condition parameter="RECOGNIZED_ARUCO_MARKERS"
                 comparison="includes" value="2" />
      <action name="LAND"/>
      <termination mode="END_MISSION" />
    </event>

    <event name="Subject recognized by other drone">
      <condition parameter= "RECOGNIZED_ARUCO_MARKERS_BY_OTHERS"
                 comparison="includes" value="2" />
      <termination mode= "NEXT_TASK" />
    </event>

  </event_handling>

</mission>
```

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).

R. Brooks, “The Behaviour Language; User’s Guide,” MIT AI Lab, 1990.

A. Champandard. Understanding Behavior Trees. AiGameDev.com, 6, 2007.

Michele Colledanchise and Petter Ogren. 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, November). 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.

Downs, J., & Reichgelt, H. (1991, March). Integrating classical and reactive planning within an architecture for autonomous agents. In European Workshop on Planning (pp. 13-26). Springer Berlin Heidelberg.

Fernández Perdomo, E., Cabrera Gámez, J., Domínguez Brito, A. C., & Hernández Sosa, D. (2010). Mission specification in underwater robotics. Journal of Physical Agents, 4(1), 25-34.

David Harel. Statecharts: A visual formalism for complex systems. Science of computer programming, 8(3):231–274, 1987.

Humphrey, L., Wolff, E., & Topcu, U. (2014). Formal specification and synthesis of mission plans for unmanned aerial vehicles. In Proc. of the AAAI Spring Symposium.

Damian Isla. Handling complexity in the Halo 2 AI. In Game Developers Conference, volume 12, 2005.

Andreas Klöckner. Behavior trees for UAV mission management. In Matthias Horbach, editor, INFORMATIK 2013 volume P-220 of GI-Edition-Lecture Notes in Informatics(LNI) Proceedings, pages57–68, Koblenz, Germany,16-20 September 2013.

Andreas Klöckner, F van der Linden, and D Zimmer. The modelica behav- iortrees library: Mission planning in continuous-time for unmanned aircraft. In Proceedings of the 10th International Modelica Conference, number 96, pages 727–736, 2014.

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

Kurt, A., & Özgüner, Ü. (2013). Hierarchical finite state machines for autonomous mobile systems. Control Engineering Practice, 21(2), 184-194.

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

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

Alejandro Marzinotto, Michele Colledanchise, Christian Smith, and Petter Ögren. Towards a unified behavior trees framework for robot control. In Robotics and Automation (ICRA), 2014 IEEE International Conference on, pages 5420–5427. IEEE, 2014.

Michael Mateas and Andrew Stern. Façade: An experiment in building a fully- realized interactive drama. In Game Developers Conference, volume 2, 2003.

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.

Mueller, E. T. (2004). A Tool for Satisfiability-Based Commonsense Reasoning in the Event Calculus. In FLAIRS Conference (Vol. 4).

Mueller, E. T. (2014). Commonsense reasoning: an event calculus based approach. Morgan Kaufmann.

Nicolescu, M. N., & Matarić, M. J. (2002, July). 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.

Olsson, M. (2016). Behavior Trees for decision-making in Autonomous Driving. Master’s Thesis KTH Royal Institute of Technology, Stockholm.

Petter Ögren. 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.

P. Ridao, J. Yuh, J. Batlle, and K. Sugihara. 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, 2005.

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

G. N. Roberts, R. Sutton, and R. Allen. Guidance and control of underwater vehicles. Elsevier Science and Technology, IFAC Proceedings Volumes(1):1–40, 2003.

Rothwell, C., Eggert, A., Patzek, M. J., Bearden, G., Calhoun, G. L., & Humphrey, L. R. (2013). Human-computer interface concepts for verifiable mission specification, planning, and management. In AIAA Infotech@ Aerospace (I@ A) Conference (p. 4804).

J. L. Sanchez-Lopez , M. Molina, H. Bavle, C. Sampedro, R. A. Suarez Fernandez, P. Campoy. A Multi-Layered Component-Based Approach for the Development of Aerial Robotic Systems: The Aerostack Framework . Journal of Intelligent & Robotic Systems (in press).

Jose Luis Sanchez-Lopez, Ramon A. Suarez Fernandez, Hriday Bavle, Carlos Sampedro, Martin Molina, Jesus Pestana, and Pascual Campoy (2016): "AEROSTACK: An Architecture and Open-Source Software Framework for Aerial Robotics". The 2016 International Conference on Unmanned Aircraft Systems ICUAS 2016. Arlington, VA, USA.

Santamaria, E., Royo, P., Barrado, C., Pastor, E., López, J., & Prats, X. (2008, August). Mission aware flight planning for unmanned aerial systems. In Proceedings of AIAA Guidance, Navigation, and Control Conference and Exhibit, Honolulu (HI).

B. Schwartz, L. N¨agele, A. Angerer, and B. A. MacDonald, "Towards a Graphical Language for Quadrotor Missions," in Workshop on Domain-Specific Languages and models for Robotic systems, 2014.

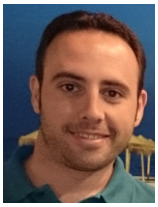
Simmons, R., & Apfelbaum, D. (1998). A task description language for robot control. In Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on (Vol. 3, pp. 1931-1937).

Yannakakis, M. (2000). Hierarchical state machines. In IFIP International Conference on Theoretical Computer Science (pp. 315-330). Springer Berlin Heidelberg.

About the authors

Martin Molina (Email: martin.molina@upm.es) is a professor at the Department of Artificial Intelligence, Technical University of Madrid since 1994 (full professor since 2012). He received his Ph.D. in computer science and artificial intelligence in 1993 and his licentiate degree in information technology in 1990 from the Technical University of Madrid (UPM). He was a visiting researcher for approximately 3 years in several research centers in the USA (AT&T Labs–Research, Stanford University and University of California–Irvine). Martin Molina led a research group about intelligent systems at his university (1999-2016). Currently, Martin Molina is a member of the research group Computer Vision and Aerial Robotics at UPM, where he leads research lines related to artificial intelligence methods applied to aerial robotics (e.g., cognitive architectures, human-machine interaction, agent-based architectures, and machine learning). He has authored more than 90 publications related to artificial intelligence, and he has more than 20 years of experience working with intelligent systems and their practical applications with an important relation with private companies related to engineering problems (transport, hydrology, aeronautics, etc.). He has been the principal investigator of 23 national research projects in the field of artificial intelligence and has participated in 7 research projects funded by the European Union (as leader or member of the research group).

Ramón A. Suárez-Fernández. (Email: suarez.ramon@gmail.com). After completing his bachelor's degree in Electrical Engineering from the University of Puerto Rico specializing in Control Systems, Ramón A. Suárez Fernández has received the Master's degree in Electronic Engineering from the University of Zaragoza and is currently pursuing a Ph. D. In Automation and Robotics from the Technical University of Madrid. As a member of the Computer Vision and Aerial Robotics Group in the Center for Automation and Robotics his research is mainly directed towards aerial and underwater robot Control Systems.



Carlos Sampedro. (Email: carlos.sampedro@upm.es) Carlos Sampedro received the Master's degree in Automation and Robotics from the Technical University of Madrid, Madrid, Spain, in July 2014. He is currently working toward the Ph.D. degree from the Computer Vision and Aerial Robotics Lab, Centre for Automation and Robotics (UPM-CSIC).His research interests include object detection and recognition using machine learning techniques, deep learning, image processing and unmanned aerial vehicles. Mr. Sampedro has received a pre-doctoral grant from the Technical University of Madrid in January 2017.

Jose Luis Sanchez-Lopez (Email: jl.sanchez@upm.es). Jose Luis Sanchez-Lopez is a post-doctoral researcher at the Computer Vision and Aerial Robotics research group of the Centre for Automation and Robotics, CSIC-UPM, since 2009 (postdoc since May 2017). He received his Ph. D. in Robotics (May 2017), his Master degree in Automation and Robotics (Oct. 2012), and his Engineering degree in Industrial Engineering (Sep. 2010), at the Technical University of Madrid. He was a visiting researcher during six months (Jul. – Dec. 2012) at Arizona State University (AZ, USA), and during thirteen months (Sep. – Dec. 2014 & Nov. 2015 – Oct. 2016) at LAAS-CNRS (Toulouse, France). His main research goal is to provide robots with the maximum level of autonomy allowing them to perform different missions without human intervention, with a special focus on aerial robots. His research interests comprise intelligent and cognitive system architectures, multi-agent systems, sensor fusion and state estimation, localization and mapping, trajectory planning, computer vision and machine learning. He has authored more than 35 publications related to these fields.

Pascual Campoy (Email: pascual.campoy@upm.es) is Full Professor on Automatics at the Universidad Politécnica Madrid UPM (Spain) and visiting professor in TUDelft (The Netherlands), he has also been visiting professor at Tong Ji University (Shanghai-China) and QUT (Australia). He currently lectures on Control, Machine Learning and Computer Vision. He is leading the Research Group on "Computer Vision and Aerial Robotics" at UPM within the Centre of Automatics and Robotics (CAR), whose activities are aimed at increasing the autonomy of the Unmanned Aerial Vehicles (UAV) by exploiting the powerful sensor of Vision, by using cutting-edge technologies in Image Processing, Control and Artificial Intelligence. He has been head director of over 40 R&D projects, including R&D European projects, national R&D projects and over 25 technological transfer projects directly contracted with the industry. He is author of over 180 international scientific publications and nine patents, three of them registered internationally. He is awarded several international prices in UAV competitions: IMAV12, IMAV13 y IARC14.