Automatic Generation of detailed Flight Plans from High-level Mission Descriptions

Davide Di Ruscio[1], Ivano Malavolta[2], Patrizio Pelliccione[1,3], and Massimo Tivoli[1]
[1]University of L'Aquila, DISIM (Italy)
[2]Vrije Universiteit Amsterdam (The Netherlands)
[3]Chalmers University of Technology | University of Gothenburg (Sweden)
davide.diruscio@univaq.it, i.malavolta@vu.nl, patrizio.pelliccione@gu.se, massimo.tivoli@univaq.it

ABSTRACT

Drones are increasingly popular since they promise to simplify a myriad of everyday tasks. Currently vendors provide low-level APIs and basic primitives to program drones, making mission development a task-specific and error-prone activity. As a consequence, current approaches are affordable only for users that have a strong technical expertise. Then, it emerges the need for software engineering techniques supporting the definition, development, and realization of missions involving swarms of autonomous drones while guaranteeing the safety today's users expect. In this paper we consider mission specifications expressed through a domain-specific modeling language which can be effectively used by end-users with no technical expertise, e.g., firefighters and rescue workers. Our generation method automatically derives the lower level logic that each drone must perform to accomplish the specified mission, prevents collisions between drones and obstacles, and ensures the preservation of no-fly zones.

1. INTRODUCTION

The next future will be pervaded by drones performing a variety of tasks in the context of civilian missions [28], like damage assessment after earthquakes, searching for survivors after airplane accidents and disasters, coastal surveillance, securing large public events, monitoring oil and gas pipelines, observing traffic flows, monitoring pollution emission, and protection of water resources. However, at the state of the art, on-site operators must deeply know all the types of used drones in terms of, e.g., flight dynamics and hardware capabilities in order to correctly operate with them. On-site operators have to simultaneously control a large number of drones during the mission execution. Moreover, professional use of drones often is realized by allocating two operators for each drone: the first controlling the movements of the drone, the second controlling the instrumentation, like photo camera and engine used to move the photo camera.

Vendors provide low-level APIs and basic primitives to program drones, thus making mission development an error-prone activity. As clearly stated in the Robotics 2020 - Multi-Annual Roadmap For Robotics in Europe[1]: "*Usually there are no system development processes (highlighted by a lack of overall architectural models and methods). This results in the need for craftsmanship in building robotic systems instead of following established engineering processes"*. Moreover, tasks are very specific and this limits the possibilities for their reuse across missions and organizations. As a consequence, current approaches are affordable only for users that have a strong expertise in the dynamics and technical characteristics of the used drones.

Entities giving permissions to fly will more and more ask for certifications about both hardware and software. Then, it emerges the need for software engineering approaches and methodologies able to support the definition, the development, and the realization of missions involving swarms of autonomous drones while guaranteeing the safety today's users expect.

This paper focuses on the definition of missions for a team of drones via a domain-specific modeling language and on the generation of low-level instructions for each drone in the swarm.

The approach builds on top of the FLYAQ platform [6, 10][2] that makes possible the specification of missions for end-users with expertise neither in ICT nor in drones dynamics, e.g., fire-fighters and rescue workers. The work in [6] explains how the different stakeholders can use the FLYAQ platform and the main benefits related to the adoption of the FLYAQ tool by organizations that need to carry out dangerous and difficult missions. Instead this paper presents the overall FLYAQ approach in terms of its domain-specific languages, their formalization, and the model transformation among them. Moreover, this paper focuses on the extensibility of the high-level DSL used to describe missions; this extensibility enables stakeholders to customize the language according to their needs. Starting from a high-level description of the mission, automatic transformations enable the automatic generation of a detailed flight plans for a team of drones. The automatic transformation guarantees that the produced detailed flight plans will satisfy the specified mission, will prevent collisions with other drones and obstacles, and will respect the specified no-fly zones.

The paper describes also the Software-In-The-Loop (SITL) simulation stack and finally the approach is illustrated by means of a real-world public event monitoring scenario.

Paper structure: Section 2 presents characteristics of drones and introduces FLYAQ. Section 3 describes the approach, Section 4 explains how it is implemented, and Section 5 discusses the key properties of the approach and their evaluation. Section 7 concludes with final remarks and future research perspectives.

WOODSTOCK '97 El Paso, Texas USA

DOI: 10.475/123_4

[1]http://sparc-robotics.eu/roadmap/
[2]http://www.flyaq.it

2. SETTING THE CONTEXT

Drones are classified in the family of UAV (Unmanned Aerial Vehicle), e.g., drones without a human pilot on board that can be either controlled autonomously by computers in the vehicle, or under the remote control of a pilot on the ground or in another vehicle. As already introduced, nowadays, mission specification and development is a difficult task, already when considering a single drone, and it becomes even more complex when dealing with missions involving a swarm of drones. In this paper we build on top of the FLYAQ [10] platform that allows non-technical operators to straightforwardly define civilian missions of swarms of flying drones at a high level of abstraction, thus hiding the complexity of the low-level and flight dynamics-related information of the drones.

FLYAQ ensures a strong adherence with the application domain by providing an extensible domain specific language, called Monitoring Mission Language (MML) that permits to graphically define civilian missions. Extension mechanisms of the language allow domain experts to specialize MML with additional tasks that are specifically tailored to the considered domain. For example, if operators are interested on monitoring solar panel installations in a rural environment, the language might be extended with tasks representing the concept of, e.g., solar panel groups, thermal image acquisition, solar panel damage discovery and notification, as well as with actions that are specific to the task. As shown in Figure 1, MML is composed of three layers, the layer to specify the *mission* through the constructs of the language, the layer to specify the *context* in which the swarm of drones has to operate, such as no-fly zones, obstacles, etc., and a *map* representing the geographical zone where the mission will be executed.

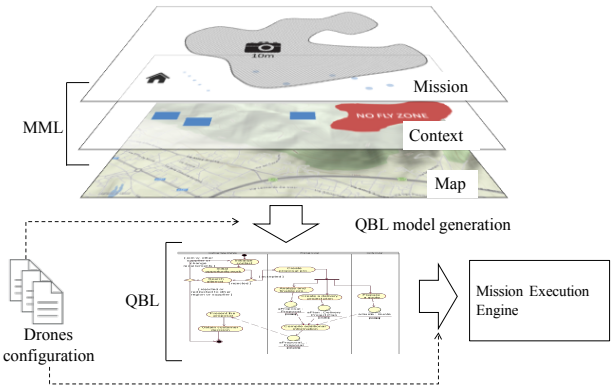

Figure 1: Overview of the FLYAQ platform

Figure 2 shows an example of mission to monitor a large public event in a small city for security reasons (tasks of the mission have been graphically manipulated so to improve the readability of the figure). From the point of view of the end user, a FLYAQ mission essentially results in a set of geographical areas, movement strategies that drones involved in the mission should perform on selected areas, such as coverage, search for an object, etc. and actions to be performed while traversing the interested waypoints, e.g., taking a picture or performing a video. The specified mission is composed of two tasks to be performed in parallel:

- *Photo Grid Task (PGT)* - this task is performed above a square (see the rectangle in Figure 2 within the circle PGT) to monitor it. The photo grid task identifies a virtual grid within the area, each cell of the grid having a size of 10 meters. The drones executing the task will fly over each cell of the grid at an altitude of 25 meters, and then will take a picture of the area directly below them.
- *Road Task (RT)* - this task refers to a polyline corresponding to the streets to be monitored (see the polyline in Figure 2 identified by the circle RT). Drones are required to fly along the polyline at an altitude of 25 meters, and take a picture every 200 meters along the polyline.

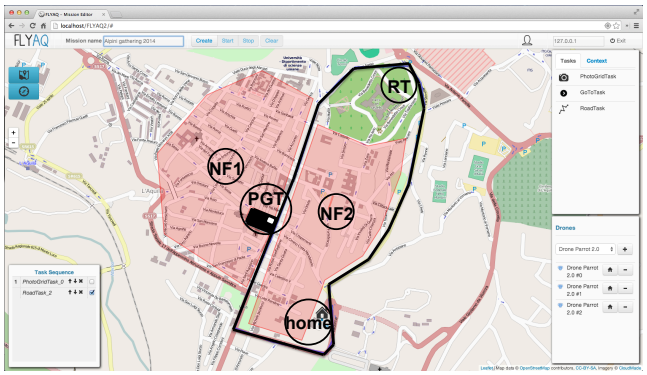

Figure 2: A public event monitoring mission

The mission will be realized by three drones that will be positioned in a large parking close to the city center (see the home circle in Figure 2): two drones will take care of executing PGT, whereas a single drone will execute RT independently. The FLYAQ platform allows the user to define also contextual information about the mission. In this example, the context specification contains two no-fly zones and an obstacle. The two no-fly zones, called NF1 and NF2, are the ones within the city center. The obstacle is within the area involved in the PGT task and it represents the area reserved by personnel of televisions to record activities performed during the event in the main square of the city. The obstacle is not very visible due to the dimension of the figure: it will be reconsidered in Section 3.2 (see also Figure 6). It will be carefully analysed when calculating movements of the drones assigned to this task.

Once the mission has been specified, then waypoints and trajectories have to be calculated. They are represented in an intermediate language called Quadrotor Behaviour Language (QBL). Examples of QBL actions include: land, take off, hover, head to, goto, read from a sensor, send feedback to the ground station, send/receive a notification to/from other drones, etc. QBL has been defined through an iterative process involving experts of the domain within the FLYAQ project. The QBL model is the input to a set of software controllers, each of them commanding a single drone, depending on the various movements and actions contained into the QBL model. Each controller is dedicated to a specific type of drone so that it is able to account for the specific flight dynamics and other characteristics of the managed drone. This aspect of the FLYAQ platform is not in the scope of this paper, so in the following we will not focus on the hardware and low-level features of the drones involved in our missions.

Limitations of FLYAQ: Currently, the FLYAQ platform does not provide any support for automatically translating the MML model into a QBL model that defines the behaviour of each drone belonging to the swarm. That is, once extending MML, the implementation of the transformation from MML to QBL is completely demanded to the platform extender. In fact, currently an extension of MML consists of (i) tasks that specialize a generic task of FLYAQ according to the domain needs, including graphical representation of the task, and (ii) algorithms to translate the task into elementary operations in QBL. Unfortunately the translation might become very complex and it is intrinsically error-prone due to the

large amount of information to be considered.

Contribution of this paper: The generation approach presented in this paper automates the translation between MML and QBL, so to extremely facilitate the extension of MML. The generated QBL specification prevents collisions among drones and between drones and obstacles, and avoids to traverse no-fly zones. Moreover, communications and interactions between drones are completely controlled so to avoid unexpected behaviours that may emerge from the collaboration of independent entities.

3. APPROACH DESCRIPTION

As shown in Figure 3, the QBL model automatically generated out of the MML one is organized into n parts, each of them devoted to describe the behaviour of a specific drone. The behaviour of a drone can be abstracted as a finite state transition system where each transition corresponds to a QBL operation. Both cyclic and alternative behaviours can be performed. Drones can exchange synchronization and communication messages (see dashed arrows in the figure). We assume that at the beginning of the mission each drone will be parked in its home location and that at the end of the mission it will land at a specific location. Then, the finite state system is structured in three parts:

- *Mission entering* consisting of the operations required to start the mission, e.g., take-off;
- *Mission tasks execution* consisting of the operations required to accomplish each mission-specific task, e.g., searching for an object in an area, taking a picture in the waypoints of an area, detecting the presence of carbon dioxide in a specific point;
- *Mission leaving* consisting of the operations required to conclude the mission, e.g., going back to home, landing.

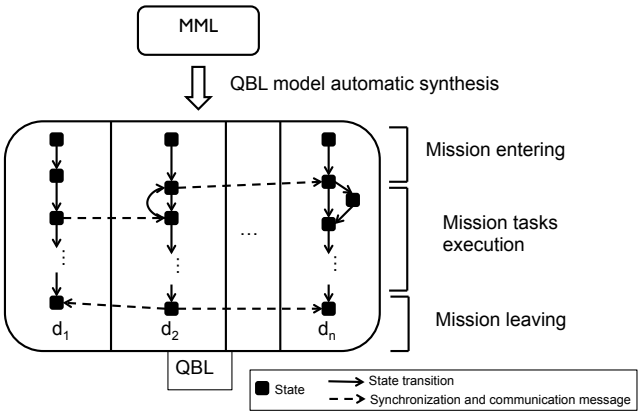

Figure 3: Overview of the approach

The main difficulty of the approach resides on the fact that MML is extensible and hence it is not possible to define once for all rules that translate MML tasks into QBL operations. Therefore, the automated generation is based on three main concepts: (i) typology and characteristics of the zone that is affected by the mission, (ii) strategies to be applied to calculate the concrete movements that drones have to perform, and finally (iii) actions to be performed while visiting the identified waypoints. As better detailed in Sections 3.3 and 3.4, these concepts pose constraints and permit to use consolidated and optimised algorithms (e.g., shorter-path calculation between two points, path planning, etc.) that will be exploited by the generation process to, e.g., determine how a drone should visit a fly-zone according to specific path planning policies. Then, a platform extender can define concrete tasks for the considered domain by associating these tasks to the general concepts defined in the platform. In this way, as described in details in the next section, it is possible to completely automate the generation of the drone behaviour specification.

3.1 MML formalization

Let $Coord$ be the universal set of geographical coordinates; for $c \in Coord$, $c.x$ denotes the longitude of c, $c.y$ is the latitude, and $c.z$ is the altitude. In MML, geometries are used to represent specific areas within the overall area of the mission. For example, a specific area can be an obstacle, a no-fly zone, or the area in which a specific task must be performed. Figure 4 shows the types of geometries supported by MML: *point*, *line*, *polygon*, *volume*[3].

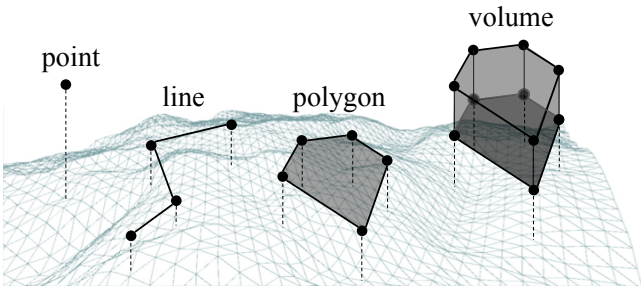

Figure 4: Geometries supported by MML

A point is an element of $Coord$. A line is an element of $Coord^n$ (n>1). A polygon is an element of $Coord^n$ where the first and last points are the same; for the sake of simplicity we assume that for each point p of a polygon, $p.z$ is the same, that is, all the points are at the same height. A volume is an element of $Coord^n \times \mathcal{R}^+$, where the second element of the pair is the height in meters. In other words, in FLYAQ a volume can be seen as polygon representing its base, with an extrusion of the height in meters. Hereafter, when the height is set to ∞, the volume represents a *no-fly zone*. _ denotes the *unspecified coordinate*, i.e., it does not correspond to any point. $\mathcal{P}$ (_ included), $\mathcal{L}$, $\mathcal{A}$, $\mathcal{V}$ and $\mathcal{Z}$ are the universal sets of points, lines, polygons, volumes and no-fly zones, respectively.

DEFINITION 1 (CONTEXT). *A* context $C=(NZ, O)$ *is a pair where* $NZ \in \mathcal{Z}^n$ (n>0) *is a list of no-fly zones, and* $O \in \mathcal{V}^m$ (m>0) *is a list of volumes representing obstacles.*

The mission, as specified in MML, is composed of a series of tasks. A task involves some drones of the available swarm. For technical reasons, we can have particular tasks that do not involve any drone, e.g., the initial and final tasks. However, these particular tasks are not necessarily required to be specified by the user, i.e., if not specified, they are automatically generated. Tasks are partially ordered, as defined by three operators: *sequencing*, one task can start only after the completion of another task, *fork*, after the completion of a task two (or more) parallel sequences of tasks start, and *join*, after the completion of two (or more) tasks that are executed in parallel, a further task will be executed. This partial ordering of tasks can be represented as a *task dependency graph*.

In the following we focus on the concept of task, see Definition 2. Informally, a concrete task specifies (i) the *movements* that

[3]Geometric and geographical concepts of the MML language are inspired from well-known OGC standards (e.g., KML): http://www.opengeospatial.org.

the involved drones have to perform in a specific zone (point, line, polygon, and volume) according to a specific *strategy* (e.g., searching for an object, covering the area with respect to a specified grid of points), and (ii) the *actions* to be performed in the traversed waypoints (e.g., taking a picture, making a video). MML defines three possible strategies. The rationale behind the following strategies is to constraint the specification of missions so to allow *automatic translation* of MML into QBL elementary operations.
$sweep(d)$: a fly-zone is fully covered with respect to a grid, whose cell dimension (in meters) is $d \in \mathcal{R}^+$.
$search(target, d)$: the visit is performed with respect to a grid, whose dimension is $d \in \mathcal{R}^+$, towards the discovery of the $target$ object. In MML, the type of $target$ is generic meaning that it must be specialized depending on the specific target of interest. For instance, it can be a PNG image or a reading from an RFID tag, and in these cases the drone must have suitable capabilities, such as image recognition or RFID reading. $target$ can be unspecified, meaning that its recognition can be delegated to an operator observing the mission through a video sent by the drone to the ground station.
$track(m, m', d)$: it is like $sweep(d)$, where the visit either starts again upon receiving the message m or ends upon receiving the message m'. m and m' are sent by the ground station. In MML, also the type of messages is generic hence requiring to specialize it depending on the application domain of the mission.

Let STR be the universal set of strategies, and let ACT be the universal set of low level functions that realize concrete drone actions (like taking a picture with a given resolution, detecting the presence of carbon dioxide, etc.). $\perp \in ACT$ denotes the *null action* meaning that it is ineffectual. A task is defined as follows.

DEFINITION 2 (TASK). *A* task $t=(M, A)$ *is a pair, where:*
- $M \in (\mathcal{P} \cup \mathcal{L} \cup \mathcal{A} \cup \mathcal{V}) \times STR$ specifies the movements to be performed in a zone according to a strategy.
- $A \in ACT \times \{i, c\}$ specifies the concrete action to be performed by the drones involved in the task. $\{i, c\}$ identifies if the action is instantaneous*, like taking a picture, or* continuous*, like taking a video. An instantaneous action is executed for each waypoint. A continuous action is started at the beginning of the task and terminated at the end. In case of $\perp$, the continuous/instantaneous flag does not matter.*

It is important to note that the concrete action A of a task t is generic, and it may correspond to a set of more specific operations to be performed by the controller of the drone. For instance, an MML action aimed at collecting some data can correspond to two operations at the controller level to be performed in sequence, e.g., an operation for taking a picture using the on-board camera and an operation for sensing the CO_2 level from a CO_2 hardware sensor. When the FLYAQ platform extender defines a new MML action, he/she has to provide suitable code for translating the action into its corresponding low-level operations in the QBL behavioural specification. Also, the drone models describing the low-level configuration of each type of used drone must define all the lower level operations that each drone is able to perform (e.g., taking pictures, sensing environmental data, etc.). Those operations must be suitably implemented and managed by the controller commanding the used drones in the field.

Hereafter, for a tuple τ, τ_i denotes i-th element of τ. $|\tau|$ is the length of τ. Let `Drone` be the universal set of drones, `home` denotes the tuple of coordinates identifying the starting (and even ending) point of the drones. `Task` denotes the universal set of tasks. With $\leq$ we denote a partial order of tasks such that $t_i \leq t_j$ means that the task t_i is performed before the task t_j. We make use of a function $\texttt{prec} : \texttt{Task} \longrightarrow 2^{\texttt{Task}}$ such that $\texttt{prec}(t_j) = \{t_k | k \neq j \wedge t_k \leq t_j \wedge \nexists h \neq k, j. t_k \leq t_h \leq t_j\}$. That is $\texttt{prec}(t_j)$ is the set of tasks that immediately precede t_j.

DEFINITION 3 (SWARM). *A swarm* $S = \{(d_1, \{t_1^1, \cdots, t_n^1\}), \cdots, (d_m, \{t_1^m, \cdots, t_k^m\})\} \in 2^{(\texttt{Drone} \times 2^{\texttt{Task}})}$ *is a set representing the drones that are involved in some mission's tasks.*

For automatic generation purposes, a drone cannot be involved in tasks belonging to different parallel flows. This guarantees that the tasks in which a drone is involved can be totally ordered with respect to $\leq$. By abusing notation, for a drone d, $S(d)$ denotes the tuple of tasks that involve d. The tasks in $S(d)$ are ordered with respect to $\leq$. Thus, $S(d)_1$ denotes the minimum task in $S(d)$. For a task t, $S(t)$ denotes the (possibly empty) set of drones involved in t.

DEFINITION 4 (TASK DEPENDENCY GRAPH). *A* task dependency graph $G = (T, D)$ *is a pair where:*
- $T \subseteq \texttt{Task}^n$ is a list of tasks ordered with respect to $\leq$.
- $D = D_{seq} \cup D_{fork} \cup D_{join}$, where:

- *$D_{seq} \subseteq \texttt{Task} \times \texttt{Task}$ is the set of transitions from a task to another one. It expresses the ordering imposed by the sequencing operator.*
- *$D_{fork} \subseteq \texttt{Task} \times 2^{\texttt{Task}}$ is the set of transitions from a task to a set of tasks passing through a fork operator.*
- *$D_{join} \subseteq 2^{\texttt{Task}} \times \texttt{Task}$ is the set of transitions from a set of tasks to a task passing through a join operator.*

DEFINITION 5 (MISSION). *A* mission $M = (C, S, G)$ *is a tuple where C is the context, S is the swarm, and G is the task dependency graph.*

3.2 QBL formalization

Beyond moving and performing actions, a drone handles a n-sized list of message queues, where n is the number of drones involved in the mission. For a drone d_i, the j-th entry of the list (with $j \neq i$) is the queue storing the messages received by the drone d_j. The i-th entry of the list is used to store the messages received by the ground station. `MSG` is the universal set of messages.
QBL defines seven elementary operations:
- `NoOp` represents the *null operation* meaning that it is always ineffectual, i.e., it does not produce any movement.
- $\texttt{TakeOff}(h)$ realizes the take off at the altitude $h \in \mathcal{R}^+$ expressed in meters. $\texttt{TakeOff}(0)$ corresponds to `NoOp`.
- `Land` realizes the landing.
- $\texttt{Hover}(\texttt{Msgs})$ represents the hovering until all the messages in $\texttt{Msgs} \in 2^{\texttt{MSG}}$ have been received. If $\texttt{Msgs} = \emptyset$, then the execution of `Hover` corresponds to `NoOp`.
- $\texttt{HeadTo}(\alpha)$ represents the rotation of an angle $\alpha \in \mathcal{N}$ with respect to the North.
- $\texttt{Goto}(p)$ is the movement that permits to reach $p \in \texttt{Coord}$.
- $\texttt{Notify}(\texttt{drones}, m)$ represents the multicast sending of $m \in \texttt{MSG}$ to all drones in $\texttt{drones} \in 2^{\texttt{Drone}}$. $\texttt{Notify}(\emptyset, m)$ corresponds to `NoOp`.

Let PL be the universal set of propositional logic formulae, $\texttt{t} \in PL$ (resp., $\texttt{f} \in PL$) denotes the Boolean value $true$ (resp., $false$). We will also denote with $\mathcal{B}$ the set $\{\texttt{t}, \texttt{f}\}$.

DEFINITION 6 (DRONE BEHAVIOUR SPECIFICATION). *A* Drone Behaviour Specification *for $d_i \in \texttt{Drone}$ is a tuple $DBS^i = (S^i, OP^i, \Delta^i, s_0^i, s_f^i)$, where:*
- S^i is the set of states.
- OP^i is the set of elementary operations.
- $\Delta^i \subseteq S^i \times OP^i \times S^i \times ACT \times \mathcal{B} \times PL \times PL$ is the transition function, and $(s, op, s', act, flag, \phi_{op}, \phi_{act}) \in \Delta^i$ is a transition:

- *s (resp., s') is the source (resp., target) state;*
- *op is the performed elementary operation;*
- *act is the concrete action to be performed after the execution of op;*
- *if $flag$ is* `t` *then the transition from one movement to another will be* fluid*, i.e., without requiring decreasing the velocity at the end of a movement before initiating the next one, if it is* `f` *the transition from one movement to another will be* non-fluid *and this will cause a stop after the execution of the movement;*
- *ϕ_{op} (resp., ϕ_{act}) is a predicate in PL representing the condition that must be evaluated during the execution of the mission to determine if op (resp., act) must be executed (in case it is evaluated to* `t`*) or not (if evaluated to* `f`*). If op=$NoOp$ (resp., act=$\perp$) it does not matter whether ϕ_{op} (resp., ϕ_{act}) is* `t` *or* `f`*, i.e., no operation (resp., action) will be performed.*

- $s_0^i \in S^i$ is the initial state of drone d_i.
- $s_f^i \in S^i$ is the final state of drone d_i.

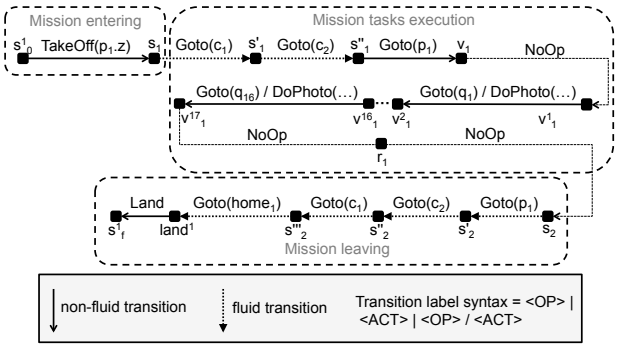

Figure 5: Drone behaviour specification for d_1

Figure 5 shows the transition system corresponding to DBS^1 for drone d_1 of the scenario in Section 2. As described in Section 3, the behaviour of d_1 consists of three phases:

Mission entering: This phase concerns the mission entering (see the transition from s_0^1 to s_1 in Figure 5): d_1 takes off from `home` and reaches the altitude of the mission's starting point.

Mission tasks execution: This phase concerns the execution of each task in which d_1 is involved. d_1 is involved in only one task, i.e., task t_1. Since t_1 involves both d_1 and d_2, d_1 has to monitor half of it (see the area delimited by p_1, p_2, q_7, and q_{10}) by discretizing its sub-area according to the grid of points shown in Figure 6. The other half is monitored by d_2. This avoids collisions between d_1 and d_2 during the execution of t_1. According to the specified strategy, i.e., `sweep`(10), and grid dimension, d_1 covers its discretized sub-area by performing a specific visit plan (see the sequence of arrows shown in the right-hand side of the figure). As detailed in Section 3.4 the performed visit plan is computed by executing a suitable coverage algorithm.

First, d_1 reaches the starting point of the mission by means of a sequence of fluid `GoTo` operations (except for the last `GoTo` that is non-fluid). The mission entering path is calculated in order to avoid both collisions with other drones in the mission and traversing no-fly zones. In fact, as shown in the right-hand side of Figure 6, d_1 approaches the mission by traversing the c_1 and c_2 points, hence reaching p_1, which is a vertex of the area specified for the execution of task PGT. Traversing these points means that d_1 avoids specified no-fly zones and the path specified for the execution of task RT by drone d_3, hence avoiding also collisions with d_3. Accordingly, for d_2 and d_3 different mission entering paths are calculated.

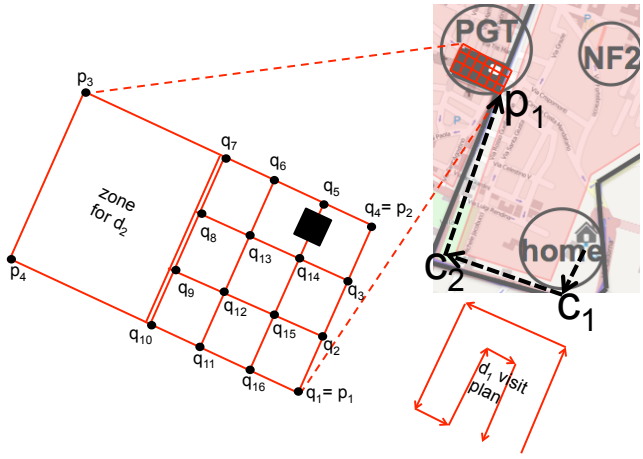

Figure 6: Reasoning to obtain the drone behaviour specification

Second, d_1 covers its sub-area by means of a sequence of `GoTo` operations and `DoPhoto` actions; the sequence of waypoints is: $q_1, \cdots, q_{16}$ (see the mission task execution part of Figure 5). Notice that q_1 coincides with p_1, then the first `GoTo`, i.e., `GoTo`(q_1), will have no effect since the d_1 will be already in p_1. This is a side effect of having the code generated, however this has no effect on the mission execution and this `GoTo` operation can be removed by minimizing the final DBS.

Mission leaving: This phase concerns exiting the mission hence leading d_1 to come back to home and land (see transitions from s_2 to s_f^1 in Figure 5).

Sections 3.3 and 3.4 provide details on how DBS^1 is automatically generated out of the MML specification. Within FLYAQ, the QBL specification is the input of a set of low-level controllers that interpret the QBL specification at run-time; this last step is not in the focus of this paper.

3.3 Automatic generation of drone behaviour specifications

The generation of DBS^i for each drone d_i is obtained through a breath-first visit of the dependency graph of the MML mission specification. In this section we focus on the explicit construction of the transition function Δ^i for a generic drone d_i. The generation of the other elements of DBS^i (i.e., states and elementary operation) is implicit with respect to the generation of Δ^i. At the beginning the initial and final states s_0^i and s_f^i are the only states in S^i and OP^i is empty. At a generic step of the synthesis process, when generating a transition $(s,op,s',act,flag,\phi_{op},\phi_{act})$ in Δ^i, s and s' are implicitly added to S^i (if not already added). Analogously, op is added to OP^i. The transition function is synthesized as the disjoint union of three different sets of transitions:

Mission Entering (ME) rule: Δ_e^i represents the QBL operations to be executed for letting d_i start the mission;

Mission Tasks Execution (MTE) rule: Δ_t^i represents the QBL operations to be executed by d_i to accomplish its assigned tasks. For each task in which d_i is involved, d_i (i) will correctly approach the starting point of the task without collisions with other drones and obstacles and without traversing no-fly zones, and (ii) it will accomplish it;

Mission Leaving (ML) rule: Δ_l^i represents the QBL operations to be executed for leaving the mission, i.e., correctly exiting from the final task by letting d_i come back to home.

At the beginning of the mission, the position of d_i is identified by $\texttt{home}_i$. Then, the position taken by d_i at the end of the execution of the ME rule is the point reached by the last Goto operation. This is the starting point for the execution of the operations synthesized by the MTE rule. The execution of the last MTE rule identifies the starting point for the ML rule. Hereafter, *cp* denotes the tuple storing, at each iteration, the current position of each drone.

Auxiliary functions: The above rules make use of three auxiliary functions that implement suitable operations to (i) distribute the geographical area of each task into a set of (sub-)areas, each of them assigned to a specific drone (see Divide below); (ii) let a drone approach the mission by reaching the starting point in the zone assigned to it (see Appr below), and (iii) cover the assigned zone according to the specified strategy and by performing the specified actions for each way point (see Cover below). Let $n>1$ be the number of drones involved in the mission, the three functions are defined as follows (details about their realization are given in Section 3.4).

- $\texttt{Divide} : \texttt{Task}\times\texttt{Coord}^n \times (\mathcal{Z}^m\times\mathcal{V}^p) \longrightarrow (\mathcal{P}\cup\mathcal{L}\cup\mathcal{A}\cup\mathcal{V})^n$ $(m,p\geq 0)$. It takes as input the specification of a MML task, the current positions of all the involved drones, and the specified context. Divide gives as output a tuple of sub-zones representing a spatial partition of the task space. The i-th area is the one assigned to drone d_i.
- $\texttt{Appr} : \texttt{Coord}\times(\mathcal{P}\cup\mathcal{L}\cup\mathcal{A}\cup\mathcal{V}) \times (\mathcal{Z}^m\times\mathcal{V}^p) \longrightarrow \mathcal{L}$ $(m,p\geq 0)$. It takes as input the current position of a drone, the mission sub-zone assigned to it (as retrieved by Divide), and the context. Appr gives as output the path that the drone has to perform for correctly reaching, from its original position, the mission starting point in the sub-space assigned to it.
- $\texttt{Cover}: \texttt{Coord}\times(\mathcal{P}\cup\mathcal{L}\cup\mathcal{A}\cup\mathcal{V})\times\mathcal{R}^+\longrightarrow(\texttt{Coord}\times\mathcal{N})^m$ $(m>0)$. It takes as input the current position of a drone involved in the task of interest, the mission sub-zone assigned to it, and the resolution of the grid used to discretize the sub-zone that the drone has to cover according to the performed visit plan. Cover gives as output a tuple of what we call *enriched points*. Each enriched point is a pair of a geographical coordinate and a rotation angle. The coordinate is used for describing Goto operations and the rotation angle is used for describing HeadTo operations due to, e.g., scenarios in which the drone has to make a picture or a video of an object.

ME rule: the drone has to take off and reach the altitude of the starting point for the execution of task $S(d_i)_1$, i.e., the initial task of the mission for d_i. The starting point is determined by using the Divide and Appr auxiliary functions.

$$\Delta_e^i=\left\{\left(s_0^i,\texttt{TakeOff}(p_1.z),s_1,\bot,\texttt{f},\texttt{t},\texttt{t}\right)\middle|\begin{array}{l}t_1=S(d_i)_1\wedge\\ l=\texttt{Divide}(t_1,\texttt{home},C)\wedge\\ p=\texttt{Appr}(\texttt{home}_i,l_i,C)\end{array}\right\}$$

MTE rule: this rule automatically generate the QBL operations to both correctly reach the starting point for the execution of task $S(d_i)_j$ and accomplish it $(1\leq j\leq|S(d_i)|)$.

For the sake of presentation, for a $p\in\texttt{Coord}^n$ (with $n\geq 1$), the notation $s_1\stackrel{p}{\Longrightarrow}v_1$ is used to represent the following sequence of transitions: $(s_1,\texttt{Goto}(p_1),s_1',\bot,\texttt{t},\texttt{t},\texttt{t}),\cdots,(s_1^{n-1},\texttt{Goto}(p_{n-1}),s_1^n,\bot,\texttt{t},\texttt{t},\texttt{t})$, $(s_1^n,\texttt{Goto}(p_n),v_1,\bot,\texttt{f},\texttt{t},\texttt{t})$. That is, it is the sequence of Goto operations that must be performed in order to move from the first point in p to the last point throughout all the intermediate points. All the movements are fluid except for the last one.

Furthermore, this rule uses a function $\texttt{enabler}: \texttt{Task}\rightarrow 2^{MSG}$ such that $\texttt{enabler}(t_h)=\{\texttt{done}_j^k \mid t_k\in\texttt{prec}(t_h)\wedge d_j\in S(t_k)\}$. $\texttt{enabler}(t_h)$ is the set of messages notifying the completion of the tasks immediately preceding the task t_h. These notifications are sent broadcast by the drones involved in the tasks preceding t_h. They inform a drone d involved in t_h that the preceding tasks have been accomplished and, hence, d can start to execute the operations required to accomplish t_h. With $e\in t_h$ we denote that e is an element of the specification of t_h.

For the sake of simplicity, we define this rule for instantaneous actions only. Its definition for continuous actions is straightforward: a continuous action is started at the first point of interest of the task and terminated at the last one. Also, for an enriched point $p\in\texttt{Coord}\times\mathcal{N}$ such that $p=(c,\alpha)$, an action $a\in ACT$, and a target object tg, the notation $s\xrightarrow{p,a,tg}s'$ is used to represent the transition $(s,\texttt{Goto}(c),s',a,a=\bot,c\neq\_\wedge\neg found(tg),\neg found(tg))$ or, alternatively, $(s,\texttt{HeadTo}(\alpha),s',a,a=\bot,c=\_\wedge\neg found(tg),\neg found(tg))$. $\neg found(tg)$ is evaluated to either t, if the target object tg is not found, or f otherwise.

If tg is not found, the enabled transition represents a single execution step that the drone has to perform to correctly cover the specified enriched point according to the specified MML strategy (sweep, target, or track). It results in an elementary operation that is either a Goto or a HeadTo movement, depending on whether a geographical coordinate or, alternatively, an angle is specified. If an action $a\neq\bot$ is specified, then the movement is not fluid and after the movement a is performed. Otherwise, the movement is fluid. If tg is found, no movement or action will be performed. This will hold for every remaining transition in the task and the next operation that will be executed is the NotifyTrack described in the following.

For a task t, $\texttt{NotifyTrack}(s'',m',s,m,s',t)$ is used to represent the following transitions: $(s,\texttt{Hover}(\{m\}),s',\bot,\texttt{f},\texttt{t},\texttt{t})$ and $(s,\texttt{Hover}(\{m'\}),s'',\bot,\texttt{f},\texttt{t},\texttt{t})$ with $track(m,m',d)\in t$, or $(s,\texttt{Hover}(\emptyset),s',\bot,\texttt{f},\texttt{t},\texttt{t})$ if t does not define any track strategy.

$$\Delta_l^i=\left\{\left(\begin{array}{c}s_j\stackrel{q}{\Longrightarrow}v_j\\ (v_j,\texttt{Hover}(\texttt{enabler}(t_j)),v_j^1,\bot,\texttt{f},\texttt{t},\texttt{t}),\\ v_j^1\xrightarrow{p_1,a,tg}v_j^2,\\ \cdots,\\ v_j^n\xrightarrow{p_n,a,tg}v_j^{n+1},\\ \texttt{NotifyTrack}(v_j^1,m',v_j^{n+1},m,r_j,t_j),\\ (r_j,\texttt{Notify}(S(t'),\texttt{done}_i^j),s_{j+1},\bot,\texttt{t},\texttt{t},\texttt{t}),\end{array}\middle|\begin{array}{l}\forall j=1,\cdots,|S(d_i)|.\\ t_j=S(d_i)_j\wedge\\ l=\texttt{Divide}(t_j,cp,C)\\ \wedge\, q=\texttt{Appr}(cp_i,l_i,C)\\ \wedge\,(\texttt{sweep}(d)\in t_j\vee\\ \texttt{search}(tg,d)\in t_j\vee\\ \texttt{track}(m,m',d)\in t_j)\\ \wedge(a,i)\in t_j\wedge\\ p=\texttt{Cover}(cp_i,l,d)\wedge\\ 1\leq n\leq|p|\end{array}\right\}$$

ML rule: this rule generates the QBL operations that are required to correctly leave the last task in $S(d_i)$, i.e., the final task of the mission for d_i, hence coming back to $home_i$.

$$\Delta_l^i=\left\{\left(\begin{array}{c}s_k\stackrel{p}{\Longrightarrow}\texttt{land}^i,\\ (\texttt{land}^i,\texttt{Land},s_f^i,\bot,\texttt{f},\texttt{t},\texttt{t})\end{array}\middle|\begin{array}{l}k=|S(d_i)|+1\wedge\\ p=\texttt{Appr}(cp_i,(\texttt{home}_i),C)\end{array}\right\}\right.$$

3.4 Auxiliary functions

In our approach auxiliary functions are based on state-of-the-art algorithms for solving problems like polygon partitioning, path finding, graph traversals, and so on [27]. The modularity of the FLYAQ platform allows a straightforward inclusion of alternative algorithms and/or future advances of existing ones without affecting the generation process. In the remainder of this section we detail the auxiliary functions.

Divide. The goal of this function is to distribute the geographical area of a task into a set of (sub-)areas, each of them assigned to

a specific drone. We assume that obstacles and no-fly areas can overlap with the geographical area of a task but, for the sake of simplicity, we assume that there is no no-fly zone crossing the geographical area and cutting it into two isolated areas. Also, to keep the `Divide` function computationally lightweight, we assign each sub-area to the drone with the smallest distance to the centroid of the sub-area in the Euclidean plane. Depending on the geometry of the geographical area of the task, we can distinguish between the following cases:

- `Point`: we can safely assume that a single drone has been assigned to it, so there is no need to divide the area.
- `Line`: being l the length of the polyline, and being n the number of drones assigned to the task, the polyline is divided into n polylines of length l/n, each of them assigned to a drone.
- `Polygon`: the first step is to construct a polygon representing the geographical area. Being p the polygon to be constructed, the vertexes of p are the coordinates of each point of the geographical area; also, for each obstacle or no-fly zone *ob* intersecting the geographical area, we add a hole to p with the same vertexes of the intersection between p and *ob*. Then, we can formulate this problem as a polygon partitioning problem over p [19, 27]. Extensive research has been performed on solving this specific problem, both theoretically [19, 27] and in more practical settings for robotic systems [2, 29, 16, 4]. In this paper we adopt the algorithm proposed in [4]. The rationale of this choice is that the algorithm employs a set of heuristics to produce partitions that in many cases appear natural and it works for arbitrary polygons, even with holes. Finally, we keep track also of the open edges of each sub-area, i.e., those edges that do not touch any edge of any other identified sub-area; they are used as "entering points" by the `Appr` function.
- `Volume`: in order to make this case manageable, we reduce this case to the two-dimensional one, thus enabling the use of the previously described algorithm [4]. In so doing we are assuming that obstacles within a volume have infinite height. This trade-off seems reasonable to us because the gains in terms of algorithm simplicity and computation demand are high. So, the result of the $Divide$ function is a set of sub-volumes, each of them having as base the corresponding polygon identified by the algorithm in [4]; all the sub-volumes have the same height of the input volume. Finally, each drone is assigned to each identified sub-volume.

Appr. This function generates the obstacle- and collision-free path that a drone d must travel to reach a geographical area a. Independently from the kind of geographical area (i.e., point, line, polygon, or volume), the `Appr` function can be formulated as a path planning problem for multiple vehicles in the three-dimensional world [27]. Path planning is still one of the open problems in the field of autonomous systems, especially as the number of degrees of freedom increases (e.g., depending on the differential constraints depending on the vehicle movements dynamics, on its minimum and maximum speed, on the presence of other vehicles moving within the environment, etc.). Tens of path planning algorithms exist, each of them applying different kinds of heuristics; an overview and deep discussion of existing path planning algorithms can be found in [13]. Since the FLYAQ platform is independent of the algorithm internally used by the *Appr* function, for the sake of simplicity in this paper we apply a basic algorithm (leaving room for further refinements). Specifically, our algorithm works according to the following steps:

1. *Obstacles enlargement*: replace each obstacle *ob* in the environment by its three-dimensional Minkowski sum with a sphere with radius r. The value of r is computed as being the sum of (i) the length of the maximum side of the drone and (ii) an arbitrary value ϵ representing a safety boundary with respect to the minimum distance that d can fly with respect to any obstacle boundary. Intuitively, this operation creates larger obstacles, defined by the shadow traced as the sphere walks a loop around each of them while maintaining contact with it [27]. This operation enables us (i) to automatically discard all passages which are too narrow for the drone and (ii) to have a minimum safety distance between the drone and each obstacle.
2. *Target points identification*: target points are those points that the drone can travel to reach the geographical area a. Target points are identified by firstly defining a set of potential target points TP, depending on the geometry of a (they will be considered in the last step of the basic algorithm). If a is a point, then TP contains only a itself; if a is a line, then TP contains both the first and last points of the polyline representing a; if a is a polygon, then TP contains every vertex of each open edge of the area a, as they have been identified by the `Divide` function; if a is a volume, firstly we consider the two polygons representing its base at the lowest and highest heights, secondly TP is computed by applying the same procedure for the case of polygons for both of them.
3. *Trajectory definition*: firstly, we compute the set SP containing the shortest paths between the current position of the drone d and all the target points in its TP set by iteratively applying a 3D extension of the well-known visibility graph algorithm [7]. We use the visibility graph algorithm since (i) it is optimal in terms of the length of the solution path, and (ii) its implementation is quite simple with respect to other path planning algorithms [26]. A classical caveat of the visibility graph algorithm is that it tends to take the robot as close as possible to obstacles, however we alleviate it by means of our obstacles enlargement preliminary step. Secondly, we select the shortest path among the paths in SP.

It is important to note that the trajectories generated by different executions of the `Appr` function may intersect. In this context, the trade-off we make is to assign a fixed altitude for each drone at each trajectory intersection, so that the drones travels each trajectory intersection at different altitudes, thus avoiding potential collisions by design. We are aware that this solution may lead to potentially inefficient flight plans; as future work we are planning to refine this part of the FLYAQ platform in order to provide a refined version of the `Appr` function that takes into consideration also the concept of time when planning the trajectories of the drones within the environment.

Cover. This function takes as input a starting position s of a drone d, a geographical area a, and a real number r representing the resolution of the grid that implicitly discretizes a. It produces a set of couples <`point`,`angle`>, where *point* is an ordered list of waypoints to be travelled by d for covering the whole area a, and `angle` is an order list (with the same length of `point`) of angles representing the specific rotation angle that the drone must have at each corresponding point; the value of `angle` actually depends on the specific high-level task. The `Cover` function can be formulated as a coverage path planning problem for robotic systems. In

the literature there is a large number of algorithms for solving this problem, each of them with specific benefits, drawbacks, and application domains (e.g., underwater robots, flying robots with strict flight dynamics, demining robots, etc.) [25, 24, 9, 1]. So, depending on the type of geometry of a, we can distinguish between the following cases: if a is a point, then the function returns a itself; if a is a line of length l, then the function return l/d points, each of them having d distance by the others. If a is a polygon, then according to the classification provided in [12], our `Cover` function can be realized via an off-line, optimal coverage algorithm with support for polygonal and non-rectilinear boundaries. In light of this, we identified the algorithm presented in [24] as a good candidate for our needs. Basically, it is based on the Boustrophedon cellular decomposition that ensures a complete coverage of the available free space, while minimizing the path of the drone within a known area; it also accounts for a fixed entry point of the robot, that will correspond to the starting point s. In order to cover the whole area at the right resolution, we fix the step size (i.e., the distance between two parallel line segments) for the Boustrophedon motion to the resolution r. The `Cover` function produces a `point` element along the identified portion of the whole motion plan at every d meters. Finally, if a is a volume, we firstly divide it horizontally into h/d planes, where h is the height of the volume, and then we iteratively apply the previously mentioned motion planning algorithm [24] for each identified plane. Also in this case we are applying a trade-off decisions between the complexity of the planning algorithm and its efficiency; we believe that this decision helps in keeping the proposed function easy to understand, without any dependency to complex 3D, full-space, planning algorithms, which are very few, even in the most recent works [12].

4. IMPLEMENTATION

The generation from MML to QBL has been implemented in a model-based setting by mainly exploiting the concepts of *models*, *meta-models*, and *model transformations*. More precisely, the outputs of the three auxiliary functions applied to the source MML model are represented as three corresponding models (see the model `Divide`, `Appr`, and `Cover`). Such models (conforming to their corresponding metamodels) are taken as input by the model transformation `MM2QBL`, which is able to generate `QBL` models out of `MML` ones. Such transformation is developed by means of the Atlas Transformation Language (ATL) [18], which is a hybrid language containing declarative and imperative constructs. A fragment of the `MM2QBL` transformation is shown in Listing 1: it consists of a header section (line 2), transformation rules (lines 14-36), and a number of helpers, which are used to navigate models and to define complex calculations on them (lines 4-12).

According to the header section the `MM2QBL` transformation takes as input four input models to generate a QBL model out of them. Helpers and rules are the constructs used to specify the transformation behaviour. Each rule defines the elements to be generated by means of target patterns (e.g., lines 32-35) that specify the instances of the target metamodel to be generated (i.e., the `DBS` metaclass of the QBL metamodel) and a set of bindings. A binding refers to a feature of the type, i.e., an attribute or a reference, and specifies an expression whose value initializes the feature.

To implement the generation of auxiliary functions, corresponding helpers have been defined. For instance, the `divide` helper in lines 4-9 is able to read the source `Divide` model and retrieves as output the sub-zones representing a spatial partition of the task space, by adhering to the definition of the `Divide` function described in Section 3.4.

Listing 1: Fragment of the MM2QBL transformation

```
1 module mml2qbl;
2 create OUT : QBL from IN : MML, IN_APPR : APPR, IN_COVER
      : COVER, IN_DIVIDE : DIVIDE;
3 ...
4 helper def : divide(task : MML!Task, positions : Sequence
      (MML!Coordinate), _context : MML!Context) : DIVIDE!
      Output =
5  DIVIDE!Mapping.allInstances()->
6   select(m | m.input.task.name = task.name and
7         thisModule.sameCoordinates(m.input.positions,
              positions) and
8         thisModule.sameContext(m.input._context,_context)
9     )->first().output;
10
11 helper def : appr(...) : Sequence(APPR!Coordinate) = ...;
12 helper def : cover(...) : Sequence(COVER!Output) =  ...;
13 ...
14 rule DroneTasks {
15  from
16   s: MML!DroneTasks
17  to
18   d: QBL!Drone (name <- s.drone.name)
19  do {
20   thisModule.MissionEntering(d);
21   thisModule.MissionTasks(d);
22   thisModule.MissioneLeaving(d);
23  }
24 }
25 ...
26 rule MissionEntering(d : QBL!Drone) {
27  using {
28   approachingPoints : Sequence(APPR!Coordinate) = ...;
29   lastApproachingPoint : APPR!Coordinate =
          approachingPoints->last();
30  }
31  to
32   t:QBL!DBS (
33    drone <- d,
34    transitionFunctions <- thisModule.
          MissionEntering_TAKEOFF(t,si,lastApproachingPoint
          .altitude) ...
35   ) ...
36 }
```

We designed `MM2QBL` so to have three main rules to manage the generation of target model fragments related to mission entering, tasks execution, and leaving (see lines 20-22). Additional rules are specified for generating specific elements of the target QBL models. For instance, the generation of target `TakeOff` elements is performed by the `MissionEntering_TAKEOFF` rule, which is called by the `MissionEntering` rule (see line 34).

Due to space limitation it is not possible to provide the reader with the full implementation of the `MM2QBL` transformation. However, the interested reader can download the current implementation of the approach from http://www.flyaq.it/synthesis.

5. EVALUATION OF GENERATED MODELS

As a first step towards a realistic assessment of the *feasibility* of our automatic generation method, we modelled missions with MML and executed generated QBL models by using a Software-In-The-Loop (SITL) simulation platform. The main characteristic of SITL simulations is that the used software stack is exactly the same as the one used in real flights; the only difference with respect to real flights is that the key low level hardware drivers (e.g., GPS sensors, accelerometers, etc.) are simulated via a dedicated software. Figure 7 shows the modules of our SITL simulation setup.

Basically, all the components of our simulation stack are open source and publicly available to the community. More specifically, the main component of our SITL simulation stack is *MAVProxy*[4], that is a developer-oriented, minimalist and extendable ground control station for any unmanned autonomous vehicle. We configured

[4]http://tridge.github.io/MAVProxy

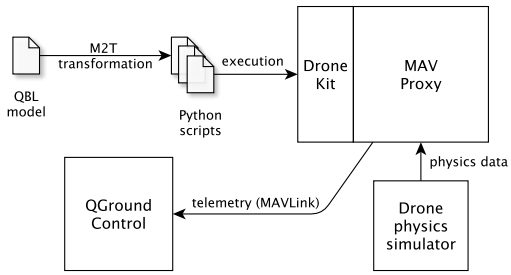

Figure 7: SITL simulation stack

MAVProxy in order to use an already existing *drone physics simulator* that is able to simulate the physical and control characteristics of the well-known Arducopter[5] drone. Moreover, the *DroneKit*[6] module within MAVProxy allows us to have programmatic access to the vehicles' telemetry and, more importantly, to have programmatic control over the vehicles' movements and operations. In light of this, we have been able to execute any QBL model by (i) automatically transforming it into a Python script for each drone defined in the modelled mission via a model-to-text transformation we developed using the Acceleo[7] template-based code generation engine, and (ii) executing those Python scripts via the DroneKit APIs. Finally, we used the QGroundControl[8] ground station to (i) seamlessly collect all telemetry data (e.g., the drones geographical positions over time, their state, battery level, performed operations) via the MAVLink communication protocol, (ii) visualize it in an interactive graphical interface, and (iii) export it as comma-separated value textual files. In turn, exported telemetry files have been analysed via a set of Java programs we specifically developed for checking whether the simulated missions actually behave according to the properties of our automatic generation method (e.g., no collisions between the drones, no violation of no-fly zones, etc.).

We exploited this simulation facility to test that generated models behave as expected. We considered missions involving 2 types of tasks (namely, for reaching a single specific geographical position and for taking photos across a grid of geographical positions) instantiated between 4 times per mission, involving 2 simulated drones and a variable number of contextual elements (mainly obstacles). When considering QBL models, in our simulations we executed a maximum of 24 actions and 30 drone movements per mission. In order to further assess the feasibility of the FLYAQ approach, we also executed a full mission in a real setting with the Parrot AR.Drone 2.0 multicopter[9]; in this case the MML model of the performed mission is composed of one drone and four tasks, each one consisting in reaching a certain geographic coordinate and taking a photo with the front camera of the drone, whereas the generated QBL model contains 6 actions and 9 movements. Interestingly, all the modelled missions have been fully executed by the FLYAQ platform without requiring to the operator neither any specific knowledge about the environment and the used drones, nor to write a single line of programming code.

Moreover, it is important to note that the automatic generation has been conceived so to guarantee properties that are important for the degree of safety that is expected by users. The overall mission is split into parts, each part assigned to a drone, and movements and actions are performed so to realize the desired mission. The outcome of the generation activity can be seen as an ordered list of waypoints for each drone involved in the mission. Waypoints, as explained before, might have associated actions to be performed, and they are traversed by following a visit plan calculated so to avoid collisions with obstacles and other drones and to respect the restrictions imposed by no-fly zones. Our method assumes that information concerning obstacles and no-fly zones is complete and correct[10]. In other words, every existing obstacle and no-fly zone is correctly modeled. A mission is composed of a set of tasks and dependencies that define a partial order between tasks. Dependency operators can be sequencing, fork, or join. To avoid concurrency issues, a drone cannot be involved in tasks belonging to different parallel flows.

Synchronization between drones is realized through communication. It is important to note that we cannot have message loses since each drone communicates with the others by directly writing to the event queue of the receiving drones (the writing action is synchronous). Ground station and drones communicate through the same mechanism. It is then easy to convince the reader that the mechanisms that realize the sequencing, fork, and join operators preserve their semantics.

Finally, it is important to note that complex interacting collective systems may expose emerging properties that represent unexpected behaviours that stem from interactions between the system parts and with the system's environment [17]. Emergent properties might be beneficial, but they can be also harmful, e.g., if they compromise the system safety. While considering critical systems it might be dangerous to accept and permit uncontrolled behaviours. Therefore, our idea is to limit as much as possible communications and interactions between drones. In particular, communications and interactions between drones are completely controlled and exclusively used for synchronizing drones before initiating new tasks. Any other form of communication between drones is blocked. This restriction seems reasonable in the context of the work of this paper. Nevertheless, in the future we will consider other more permissive solutions if we will experience their need in the practice of UAV exploitation.

6. RELATED WORK

A comprehensive survey of approaches for cooperative teams of UAV operating as *distributed processing systems* can be found in [8]. The work in [23] introduces CSL, which is a high-level feedback control language for mobile sensor networks. The style of the language is similar to that of Petri nets (missions are made of tasks with tokens and transitions). The run-time architecture of the proposed approach allows engineers to update a modelled mission at run-time by means of a patching system for the mission specification. Differently from our approach, the CSL language does not support any kind of check on the feasibility and safety of the modelled mission; also, trajectory plan in 3D is not supported.

Many *algorithms* have been proposed for automatic trajectory generation and control, with a strong focus on either trajectory optimization [15], feasibility [3], or safe obstacle and trajectories intersection avoidance [22]. The interested reader can refer to [14], which proposes an overview of existing motion planning algorithms specific for UAV guidance.

From a slightly different perspective, the work in [21] proposes a new paradigm called *cyber-physical computing cloud* (CPCC). It allows any customer to assign, check, and distribute sensing services on virtual vehicles. Essentially, this approach ports the prin-

[5]http://copter.ardupilot.com
[6]http://python.dronekit.io
[7]http://eclipse.org/acceleo
[8]http://qgroundcontrol.org
[9]http://ardrone2.parrot.com

[10]Dealing with uncertainly in this domain requires a dedicated approach; initial work might be found in [10, 11].

ciple of Platform-as-a-Service (PaaS) to the distributed robotics domain. According to this principle, the system can perform multi-customer information acquisition missions on swarms of UAV operated and maintained by a third party, similarly to how traditional web-based PaaS systems work. Differently from our approach, in [21] free-space environment is assumed and collisions are not taken into consideration. Moreover, location movements related to the tasks are manually given by the customers of the PaaS system, then tasks are assigned to physical vehicles by using a binding algorithm based on Voronoi cells.

For what concerns the activity of *mission planning and definition*, many approaches focus on the definition of (either GPS-based or vision-based) waypoints and trajectories in the real world that must be navigated by the drone in the field [5, 20].

Differently from these approaches, our main objective is to provide a software platform that makes the definition and realization of missions possible for people that are neither expert on ICT nor in robotics. In other words our platform (i) focusses on the definition of the various tasks of a monitoring mission at an higher level of abstraction, i.e., tasks and tasks dependencies; (ii) allows engineers to automatically generate detailed flight plans from a user friendly, domain-specific, and graphical description of a mission; (iii) generates flight plans that avoid obstacles, collisions and no-fly zones; (iv) does not demand to manually specify each single waypoint of the mission (that actually may be hundreds in complex missions), rather it is able to automatically compute, plan, and assign all the waypoints that must be visited by each drone of the swarm to accomplish the mission; and (v) is independent from the used task allocation, geometric and path finding algorithms, thus enabling for the use of state-of-the-art and well-established algorithms depending either on the next advances of those algorithms and on the traits of the missions to be performed in the future.

7. CONCLUSIONS AND FUTURE WORK

This paper describes an automated method to generate from a user friendly, domain-specific, and graphical description of a mission, the low level logic that each drone composing a swarm has to perform. The generation approach avoids (i) collisions among drones and between drones and obstacles, (ii) violations of no-fly zones, and (iii) unexpected behaviours that may come from the collaboration of independent drones.

As future work we plan to integrate the work with methodologies to control the mission execution at run-time. The idea is to exploit the synthesized DBSs at run-time so to force the drone to exhibit desired behaviours only [11]. As the proposed approach has been evaluated in a SITL simulation engine, we are also planning to perform a more thorough campaign of experiments, both by simulation and by executing the generated missions in real-world scenarios. Another interesting future work concerns the ability of accounting for time and resources consumption that are extremely important in this domain. This will enable, e.g, the possibility of statically checking mission end-to-end timelines, or the realization of resource-aware missions. Finally, we will investigate also the possibility of making communication between drones more flexible, so to enable also emerging behaviours; this will open challenging and futuristic scenarios where intelligent swarms of drones will decide at run-time operations to be performed in order to satisfy the goal of a (possibly evolving) mission.

8. REFERENCES

[1] E. U. Acar, H. Choset, A. A. Rizzi, P. N. Atkar, and D. Hull. Morse decompositions for coverage tasks. *The International Journal of Robotics Research*, 21(4):331–344, 2002.

[2] A. Agarwal, L. M. Hiot, N. T. Nghia, and E. M. Joo. Parallel region coverage using multiple uavs. In *Aerospace Conference, 2006 IEEE*, pages 8–pp. IEEE, 2006.

[3] F. Augugliaro, A. P. Schoellig, and R. D'Andrea. Generation of collision-free trajectories for a quadrocopter fleet: A sequential convex programming approach. In *IEEE/RSJ International Conf. on Intelligent Robots and Systems (IROS)*, pages 1917 –1922, 2012.

[4] H. Bast and S. Hert. The area partitioning problem. In *Proceedings of the 12th Canadian Conference on Computational Geometry, Fredericton, New Brunswick, Canada, August 16-19*, 2000.

[5] S. Bouabdallah and R. Siegwart. Full control of a quadrotor. In *Intl. Conf. on Intelligent Robots and Systems*, pages 153 –158, 2007.

[6] D. Bozhinoski, D. Di Ruscio, I. Malavolta, P. Pelliccione, and M. Tivoli. Flyaq: Enabling non-expert users to specify and generate missions of autonomous multicopters. In *Proceedings of ASE'15, IEEE/ACM*, 2015.

[7] M. N. Bygi and M. Ghodsi. 3D Visibility Graph. *Computational Science and its Applications, Kuala Lampur*, 2007.

[8] G. Chmaj and H. Selvaraj. Distributed processing applications for uav/drones: A survey. In H. Selvaraj, D. Zydek, and G. Chmaj, editors, *Progress in Systems Engineering*, volume 1089 of *Advances in Intelligent Systems and Computing*, pages 449–454. Springer International Publishing, 2015.

[9] H. Choset. Coverage of known spaces: the boustrophedon cellular decomposition. *Autonomous Robots*, 9(3):247–253, 2000.

[10] D. Di Ruscio, I. Malavolta, and P. Pelliccione. Engineering a platform for mission planning of autonomous and resilient quadrotors. In *Fifth International Workshop, SERENE 2013*, pages 33–47. Springer Berlin Heidelberg, LNCS, 2013.

[11] D. Di Ruscio, I. Malavolta, and P. Pelliccione. The role of parts in the system behaviour. In *Sixth International Workshop, SERENE 2014*. Springer Berlin Heidelberg, LNCS, 2014.

[12] E. Galceran and M. Carreras. A survey on coverage path planning for robotics. *Robotics and Autonomous Systems*, 61(12):1258–1276, 2013.

[13] C. Goerzen, Z. Kong, and B. Mettler. A survey of motion planning algorithms from the perspective of autonomous uav guidance. *Journal of Intelligent and Robotic Systems*, 57(1-4):65–100, 2010.

[14] C. Goerzen, Z. Kong, and B. Mettler. A survey of motion planning algorithms from the perspective of autonomous uav guidance. *Journal of Intelligent and Robotic Systems*, 57(1-4):65–100, 2010.

[15] M. Hehn and R. D'Andrea. Quadrocopter trajectory generation and control. In *IFAC World Congress*, volume 18, n. 1, pages 1485–1491, 2011.

[16] S. Hert and B. Richards. Multiple-robot motion planning = parallel processing + geometry. In *Sensor Based Intelligent Robots*, pages 195–215. Springer, 2002.

[17] C. W. Johnson. What are emergent properties and how do they affect the engineering of complex systems? *Reliability Engineering & System Safety*, 91(12):1475 – 1481, 2006. Complexity in Design and Engineering Complexity in Design and Engineering.

[18] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.

[19] J. M. Keil. Polygon decomposition. *Handbook of Computational Geometry*, 2:491–518, 2000.

[20] F. Kendoul, Y. Zhenyu, and K. Nonami. Embedded autopilot for accurate waypoint navigation and trajectory tracking: Application to miniature rotorcraft uavs. In *Intl. Conf. on Robotics and Automation*, pages 2884 –2890, may 2009.

[21] C. Krainer and C. M. Kirsch. Cyber-physical cloud computing implemented as paas. In *Proceedings of the 4th ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems*, CyPhy '14, pages 15–18, New York, NY, USA, 2014. ACM.

[22] J. Leonard, A. Savvaris, and A. Tsourdos. Towards a fully autonomous swarm of unmanned aerial vehicles. In *Control (CONTROL), 2012 UKACC International Conf. on*, pages 286 –291, sept. 2012.

[23] J. Love, J. Jariyasunant, E. Pereira, M. Zennaro, K. Hedrick, C. Kirsch, and R. Sengupta. Csl: A language to specify and re-specify mobile sensor network behaviors. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 67–76. IEEE, 2009.

[24] R. Mannadiar and I. Rekleitis. Optimal coverage of a known arbitrary environment. In *Robotics and Automation (ICRA), 2010 IEEE Int. Conf. on*, pages 5525–5530. IEEE, 2010.

[25] S.-W. Ryu, Y.-H. Lee, T.-Y. Kuc, S.-H. Ji, and Y.-S. Moon. A search and coverage algorithm for mobile robot. In *Ubiquitous Robots and Ambient Intelligence (URAI), 2011 8th International Conference on*, pages 815–821, Nov 2011.

[26] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.

[27] S. S. Skiena. The algorithm design manual, 1997. *Stony Brook, NY: Telos Pr*, 504.

[28] T. Skrzypietz. *Unmanned Aircraft Systems for Civilian Missions*. BIGS policy paper: Brandenburgisches Institut für Gesellschaft und Sicherheit. BIGS, 2012.

[29] A. Xu, C. Viriyasuthee, and I. Rekleitis. Efficient complete coverage of a known arbitrary environment with applications to aerial operations. *Autonomous Robots*, 36(4):365–381, 2014.