Mission Specification and Execution of Multidrone Systems

Markus Gutmann, Bernhard Rinner
Institute of Networked and Embedded Systems
University of Klagenfurt, Austria
markus.gutmann@aau.at, bernhard.rinner@aau.at

***Abstract*—Small unmanned aerial vehicles, commonly called drones, enable novel applications in many domains. Multidrone systems are a current key trend where several drones operate collectively as an integrated networked autonomous system to complete various missions. The specification and execution of multidrone missions are particularly challenging, since substantial expertise of the mission domain, the drone's capabilities, and the drones' software environment is required to properly encode the mission. In this position paper, we introduce a specification language for multidrone missions and describe the transcoding of its components into the multidrone execution environment for both simulations and real drones. The key features of our approach include (i) domain-independence of the mission specification, (ii) readability and ease of use, and (iii) expandability. The specification language has a simple syntax and uses a parameterized description of execution blocks and mission capabilities, which are derived from native drone functions. Domain-independence and expandability are provided by a clear separation between the specification and the implementation of the mission tasks. We demonstrate the effectiveness of our approach with a selected multidrone mission example.**

I. Introduction

Throughout the past years, small unmanned aerial vehicles (UAVs) or drones have become increasingly popular among many users ranging from hobbyists to civil authorities and industries to researchers. There is a wide variety of civilian drone applications including traffic monitoring, remote sensing, search-and-rescue operations, delivery of goods, security and surveillance, and precision agriculture [1]–[4]. While the number and types of drones are growing and use cases become more complex, the necessary software for controlling drones and their mission deployments is evolving at a slower pace. Such software support is particularly relevant for multidrone missions where the description of mission objectives and their execution should be handled efficiently and effectively.

Figure 1 depicts a search-and-rescue mission as a running example in this paper. A fleet of heterogeneous drones supports a rescue operation after a plane crash. In a nutshell, first responders need to (i) acquire an overview of the accident site, (ii) localize sources of emissive radiation, and (iii) search for victims. This mission consists of three concurrent and interdependent tasks: First, patrolling the site with two video-drones along predefined routes and constantly streaming videos to a ground control station (GCS). Second, exploring the site with flir-drones and a sensor-drone to search for victims and radiation sources, respectively. Third, in case of successful detection, load a corresponding light beacon on a beacon-drone, fly to the target position, and unload the beacon.

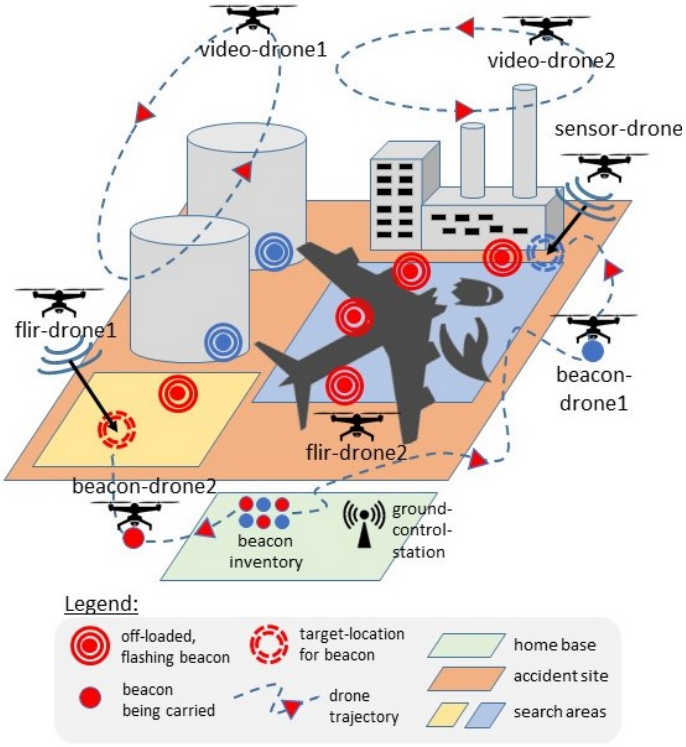

Fig. 1: Example search-and-rescue mission with heterogeneous drones. Two video-drones patrol over different areas streaming live video data, while two flir-drones search for victims and a sensor-drone searches for radiation sources. In this scenario, flir-drone1 has spotted a victim, which triggers beacon-drone2 to fly a red-light beacon to the victim. Similarly, the sensor-drone has found a radiation source that triggers beacon-drone1 to drop a blue-light beacon there.

In this position paper, we introduce a specification language for multidrone missions and describe the transcoding of its components into the multidrone execution environment for both simulations and real drones. There are already several mission specification methods for robot systems available, and they strongly vary concerning the level of specification detail, the target missions and users, and the implementation maturity. The key features of our approach include (i) domain-independence of the mission specification, (ii) readability and ease of use, and (iii) expandability. This combination is novel for multidrone systems, since existing specification approaches typically focus only on two of these features.

The remainder of this paper is organized as follows. Section II discusses related work. Section III introduces our mission specification language design. Section IV describes the envisioned architecture and used tools. Section V sketches a specification example, and Section VI concludes the paper.

II. State of the Art

There are several methods that offer user-oriented ways of specifying multidrone missions. These methods vary concerning the used language families and user interfaces as well as application domains, target platforms, and end users. Prominent representatives for procedural multi-purpose programming languages (such as Java) are the (swarm-) programming languages KARMA [5], VOLTRON [6], PaROS [7], and BUZZ [8], while representatives for declarative markup languages (such as XML) include the task-based mission specification language TML [9] and the mission description language MDL [10]. Programming-like languages provide a much richer feature set and a higher degree of expressiveness than their markup-based counterparts. However, their handling also requires programming skills, which makes them less suitable for quick applications in dynamic ad-hoc scenarios.

The end user orientation of markup-languages comes at the cost of reduced flexibility concerning the application domain and the used platforms. CommonLang [11] and the PROMIS framework [12], which enables the specification of high-level goals rather than atomic mission steps are further examples. Schwartz et al. [13] represent a mission in a flow-chart-like diagram that is easy to comprehend and simple to generate but lacks proper support for concurrency. As a further graphical approach, RobotML [14] enables the user to graphically model architecture, communications, behavior and deployment of robotic systems.

The Temporal Action Logic [15] with its abstract task specification trees or the approach by Ashley-Rollman et al. [16] that extends the logic programming language Meld are examples for formal approaches. Apparently, such notations are difficult to handle and not suited for out-of-the-box applications. Furthermore, graphical user interfaces exist, which are typically easy to use but rather tightly bound to the application they are developed for. There are several examples of such user interfaces for drones, including FlyAQ [17], the mission definition system [18], the FlightMaster software platform [19] or simply the mission planner from ArduPilot [20].

III. Mission Specification and Execution

A. Requirements

The development of our multidrone mission specification and execution is based on the following core requirements.

a) Multidrone Support: Involving multiple drones in missions is integral to our approach, and handling of an arbitrary number of drones should therefore be provided implicitly.

b) Concurrency: The specification of concurrent processes that involve multiple drones should be intuitive and easy. Such processes include starting tasks, reacting to environmental events, or communicating with each other. Explicit synchronization of concurrent processes should be hidden from the user.

c) Language Usability: Users should be able to specify new missions with little training. The mission specification should be composed by simple expressions based on an intuitive syntax. This requirement aims at domain experts as the primary user group rather than drone experts.

d) Domain Independence: The language should not be limited to a particular application domain such as inspection, surveillance, delivery of goods, communication services, or search-and-rescue.

e) Expandability: The user should be able to easily define new mission tasks based on capabilities supported by drones. These mission capabilities should transparently expand the native specification language and the predefined drone functionalities.

f) Algorithmic Decoupling: The specification and execution of a multidrone mission and the corresponding necessary control mechanisms should be clearly decoupled from low-level algorithms of individual drones. This way, the user should not be concerned about how individual drones, for example, avoid collisions while the mission progresses.

B. Overview

Our mission specification and execution system relies on the following entities. *Drones* with a set of native functionality such as launching, landing, flying-to, target detection, and video streaming serve as the basis for a mission. A drone-API provides access to the native drone functionality. To configure our system, this functionality needs to be integrated into our middle-layer-API that decouples our (high-level) mission specification from the (low-level) functionality of the considered drone models. *Mission capabilities* are built with the functions of the middle-layer-API and serve as essential building blocks for the mission specification. A *multidrone mission* is finally composed by a set of execution blocks, which may use mission capabilities, expressed in our mission specification language.

The *ground control station (GCS)* establishes the communication with the drones and acts as execution environment for a mission. It provides (i) the middle-layer-API, (ii) the user interface for loading, compiling, and controlling a mission, and (iii) the execution engine for running a mission, in particular its execution blocks. The execution blocks, interfaces, and necessary control structures are compiled into an executable program. The mission specification language and the execution architecture are described in the following subsections.

C. Specification Language

In the following, we describe key elements of our domain-specific language (DSL) for defining multidrone missions and provide its EBNF-notation (cf. EBNF-Listing 1).

⟨mission⟩ ::= `Mission` ⟨identifier⟩ `:` ⟨block⟩ (1)
⟨block⟩ ::= ⟨seqBlock⟩ | ⟨parBlock⟩ (2)
⟨seqBlock⟩ ::= `SEQUENTIAL` ⟨identifier⟩? `:` ⟨blockBody⟩ (3)
⟨parBlock⟩ ::= `PARALLEL` ⟨identifier⟩? `:` ⟨blockBody⟩ (4)
⟨blockBody⟩ ::= (⟨statement⟩ | ⟨requirement⟩ | ⟨foreachBlk⟩ | ⟨eventTrigger⟩ | ⟨eventHandler⟩ | ⟨exceptionTrigger⟩ | ⟨exceptionHandler⟩ | ⟨block⟩) + (5)
⟨statement⟩ ::= ⟨actor⟩ + ⟨capability⟩ ⟨arguments⟩ ∗ (6)
⟨requirement⟩ ::= (`STARTIF` | `WAITFOR` | `DOWHILE`) ⟨condition⟩ ⟨block⟩ (7)
⟨eventTrigger⟩ ::= `SIGNAL` ⟨event⟩ ⟨arguments⟩ ∗ (8)
⟨eventHandler⟩ ::= (`SEQUENTIAL` | `PARALLEL`) `ON` ⟨signal⟩ `:` ⟨block⟩ (9)
⟨foreachBlk⟩ ::= (`SEQUENTIAL` | `PARALLEL`) `FOR` ⟨instance⟩(`AND` ⟨instance⟩) ∗ `AS` ⟨identifier⟩(`WITH` ⟨argsList⟩)? `:` ⟨block⟩ (10)
⟨instance⟩ ::= ⟨actor⟩ (`WITH` ⟨argsList⟩) ∗ (11)
⟨arguments⟩ ::= (`WITH` ⟨argument⟩(`,` ⟨argument⟩)∗)? (12)

EBNF-Listing 1. Excerpt of key grammar elements of our mission specification language. The elements ⟨condition⟩, ⟨execptionTrigger⟩, and ⟨exceptionHandler⟩ have a similar definition but have been omitted due to space limitation.

a) Blocks: Blocks are named containers for statements or other blocks and can be used to form a hierarchical tree structure. A *sequential* block (cf. rule 3) processes its children $c_1, c_2, \ldots, c_n$ one after the other whereas c_{i+1} is started only if c_i has finished its execution. Thus, a sequential block terminates when its last child has terminated. On the other hand, a *parallel* block (cf. rule 4) starts all children at the same time and terminates when all children have finished.

b) Statements: A statement is a call of a mission capability by stating one or multiple actors (name of the drone), the name of the capability, and optional arguments (cf. rule 6).

c) Event Triggers and Event Handlers: Event triggers are denoted with the keyword **SIGNAL**, their signal name and an arbitrary number of event arguments (cf. rule 8). Event handlers are denoted with the keyword **ON**, which is prepended with either of the keywords **SEQUENTIAL** or **PARALLEL**, and followed by the signal name to handle (cf. rule 9). Event handlers are *semi-synchronous* because they (i) handle the first event that matches the event name and that occurs in the block of the handler or in one of its predecessor-blocks and because (ii) the prepending keywords **SEQUENTIAL** and **PARALLEL** determine whether the event handler is executed immediately and in parallel to the signalling actor or whether it waits until the currently executing block of the signalling actor has finished.

d) Exception Triggers and Exception Handlers: Throwing and handling exceptions behave similarly to events. By using the keywords **THROW EXCEPTION** followed by the name of the exception and none, one, or multiple arguments, all executions of the actor that threw the exception are aborted, and execution of the corresponding exception handler starts immediately. Like event handlers, exception handlers are semi-synchronous and handle only those exceptions that get thrown in the handler's block or in one of its predecessor blocks.

e) Control Flow: Requirements that influence the execution of a mission are specified with the keywords **STARTIF** (i.e., execute a statement only if a condition is true at the time of the evaluation), **WAITFOR** (i.e., wait for a certain condition to become true before starting to execute the subsequent block), and **DOWHILE** (i.e., execute the subsequent block as long the corresponding condition is true), all followed by a conditional expression and the block to execute. Furthermore, using **FOR** executes the **FOR**-block for all declared actors and arguments either in parallel or sequentially. Note, that starting the instantiated blocks of a **FOR**-block sequentially or in parallel is independent on how the enclosed execution block starts its enclosed statements.

f) Mission Capabilities: A mission specification consists of both execution blocks and mission capabilities that actors use in the mission. Mission capabilities enable one to encapsulate mission logic to facilitate its reuse (like with a method in Java), but their key feature is to enable using middle-layer-API functions which are called according to the standard syntax of calling methods (like for example in Java):

⟨API-Call⟩ ::= ⟨actor⟩ `.` ⟨API-function⟩ `(`⟨parameterValues⟩ ∗ `)`

Here, *actor* refers to the actor on whom the capability was called and *API-function* refers to a well-defined function from the middle-layer-API (cf. Section III-D).

g) Capability Control: A mission capability consists of the following top-level sections that implement its *internal* context: **REQUIREMENTS** defining conditions that determine the execution; **DEFAULTS** setting default values for the parameters if no corresponding arguments were set by the caller; **PERFORM** defining the core execution logic using language features and API-calls; and **EXCEPTIONS** defining different error situations. Additional perform-sections with an assigned identifier might be declared to define further execution logic. While the nameless perform-section starts an activity initially, named perform-sections, which are also referred to as leashes, enable the mission operator to control already running activities. Pre-defined leashes that are available on any capability are **ABORT**, **PAUSE**, and **RESUME**. A prerequisite for using leashes is to postfix the capability name within a mission statement with a "-" followed by an identifier. For example, "*flir-drone TakeOff-F1*" starts the activity by executing the capabilities' standard perform-section. From then on, using the capabilities name together with any of the available leashes, would just call the corresponding named perform-section, like *FlyTo-F1 ABORT*.

D. Execution Architecture

Figure 2 illustrates the layered architecture of our mission execution system. The *hardware layer* abstracts the available low-level drone functionality via the native-drone-API, which serves as interface to the *execution layer*. This layer provides

the conversion from native drone functionality to mission capabilities by mapping the native-drone-API to the middle-layer-API via the API-wrapper. This mapping must be implemented during the configuration of the execution architecture. The execution engine executes the compiled mission program and communicates with the drones via the available wireless network. In particular, it sends and receives drone data by invoking the correspondent functions of the native-drone-API. The *middle layer* acts as an abstract interface between the mission layer and the drones. It unifies the diverse functionalities into a well-defined API and provides mission capabilities or supports the generation of new mission capabilities by the mission operator. Finally, the *mission layer* represents the interface to the mission operator who specifies missions with execution blocks and mission capabilities.

The top three layers are implemented on the ground control station, whereas the bottom layer is deployed on the drone platforms. Thus, to adapt the architecture to different drone platforms, only the hardware layer and the interface in the execution layer needs to be changed. This is particularly relevant for multidrone simulation environments where the simulator, such as AirSim or Gazebo, corresponds to the hardware layer and hence the entire architecture can be implemented on a single computer.

IV. Implementation

A. Mission Capabilities

Mission capabilities orchestrate available drone capabilities together with a certain capability context to mission-specific activities. When a mission statement is executed, the corresponding mission capability gets instantiated by passing the contextual values of the mission statement (e.g., actors and optional argument values) to the *internal* context of the mission capability (i.e., requirements, perform-section, and its exceptions-section). This is very similar to calling a method in a language like Java, where values used as arguments of a method call get passed to its parameters when executed. Instantiating a mission capability further means passing its context to a capability handler, which is a component of the execution engine. The capability handler is in charge of executing the capability and monitoring its execution. In addition to the internal capability context, the capability handler also uses an external capability context of every capability, which primarily consists of (i) the capability control and (ii) other capabilities whose execution affects its own execution.

Technically, there is one capability handler for every actor of a mission, who is performing activities (i.e., instantiating capabilities). Every capability handler starts every capability instance as a separately executing thread that is linked to its external context to keep its execution dynamic. This linkage is efficiently implemented via bidirectional communication channels provided by our target execution language Go[1]. We take advantage of this fundamental feature for realizing the

[1]https://golang.org/

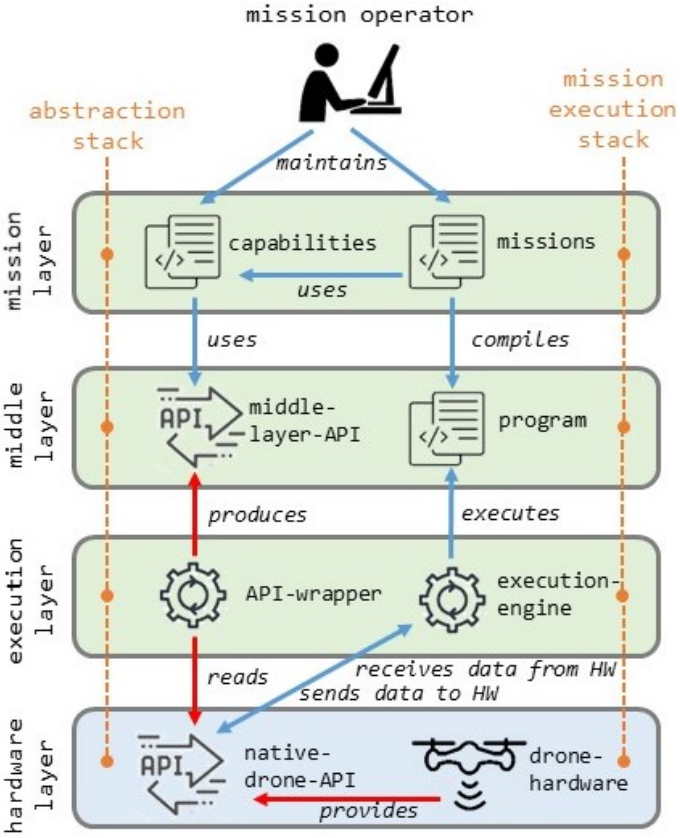

Fig. 2: Our layered mission execution architecture including the abstraction stack (left) and execution stack (right). Red arrows indicate the interactions that must be established for the configuration of the architecture. Blue arrows indicate online interactions such as the usage of mission capabilities via the middle-layer-API or the compilation and execution of a mission. The execution engine communicates with the drones via the native-drone-API.

communication that occurs during a mission (e.g., GCS-to-drone, drone-to-drone communication, and drone-internal communication where multiple running threads on (or for) the same platform exchange data. Examples for external events that affect the capability's execution are subscribed signals, handled exceptions, or simply if leashes of the capability control are called (cf. Section III-C Capability Control).

B. Mission Specification Language

We use the *Meta Programming System MPS* [21] from JetBrains for developing our mission specification language. MPS is an integrated development suite that enables and supports all aspects of developing a domain-specific language (DSL), including its design, representation, and translation into other languages. MPS utilizes so-called projectional editors at its heart to define the syntax of a language, which are used by both the developer and the user of the DSL.

In our approach, we use MPS (i) for developing our mission language, (ii) as mission-user interface to maintain the mission and mission capabilities, and (iii) for the translation of the mission specification into the high-level programming language Go. The compiled Go byte-code will eventually execute in our execution environment.

The open-source language Go enables us to efficiently realize concurrency that is key when translating our mission specification into byte-code that eventually gets executed on either different drones or the GSC. The fundamental Go features are (i) Go-routines, which resource-efficiently spawn new threads of execution, and (ii) bi-directional communication channels used to pass data between executing Go-routines. Listing 1 illustrates these two features with an example function.

```
1  func spawnDrone (ctrl chan<-string, sensor float->
       chan, string name) {
2      message := <-ctrl // receive from channel
3      sensor <- data // send into channel
4  }
5  func gcs() {
6      var controlChannels [3]string
7      var sensorChannels [3]float
8      for (int i := 0; i < 3; i++) {
9          controlChannels[i] := make(chan string)
10         sensorChannels[i] := make(chan float, 100)
11         go spawnDrone(controlChannels[i],
               sensorChannels[i], "flir-drone" + i)}
12     controlChannels[0] <- "START"
13     data := <-sensorChannels[1]}
```

Listing 1: Go-routine *spawnDrone* with input *ctrl* and output *sensor* channels (line 1). The function *gcs* (line 5) instantiates an unbuffered (line 9) and a buffered channel with a capacity of 100 messages (line 10), and passes them to *spawnDrone* in three iterations (line 11). Finally, data is sent to one of the *ctrl* channels (line 12) and received from one of the *sensor* channels (line 13).

C. Mission Execution Stack

Two code translations are necessary for transforming a mission specification into a multidrone mission execution. Initially, the mission operator writes both missions and mission capabilities using the MPS-Editors. The first translation is performed automatically by MPS, which transcodes the mission specification into Go-source code. The second translation is then the compilation of the Go-source code into byte-code of the target by the Go-installation of our execution environment.

The Go-infrastructure supports the integration of Go-code into our execution environment very well by the following three features: First, *cross-compilation* targets a specific hardware architecture, and no Go runtime environment needs to be installed on the target machines. Second, the pseudo-package *CGo* enables the Go-developer to call C-code directly from any Go-program. This is particularly significant because it eases the interaction of Go-programs with C or C++ functions which are widely used in drone platforms. Third, the *ROS/GO-client libraries* support the development on various robotic platforms. Two recent libraries are *rosgo*[2] and *goroslib*[3], which fully implement the ROS client library requirements.

V. Example Mission Specification

Listing 2 demonstrates our mission specification language with an excerpt of the example from Section I. This specification declares a mission named *Search_and_Rescue* (line 1) with one **PARALLEL** execution block named *Start_Mission* (line 2), which concurrently starts the **SEQUENTIAL** execution of two *video_drones* in blocks *Launch_Video1* (line 3) and *Launch_Video2* (line 9), and the *sensor_drone* in block *Radiation_Detection* (line 23). Since *Launch_Video1* is declared as sequential, its individual statements are processed one after the other. First, *video_drone1* takes off to a height of $20\,m$ (line 4). Once the drone has successfully reached this height, the system triggers the signal *Video1_Ready* (line 5) for whoever is asynchronously subscribed to this event. The drone then flies to the location *BuildingA* (line 6), starts there its video streaming activity with the *StartStreaming* capability (line 7), and circles around the building in a $100\,m$ radius (line 8).

```
1  MISSION Search_and_Rescue:
2    PARALLEL Start_Mission:
3      SEQUENTIAL Launch_Video1:
4        video_drone1 TakeOff WITH 20m
5        SIGNAL Video1_Ready
6        video_drone1 FlyTo WITH BuildingA
7        video_drone1 StartStreaming
8        video_drone1 FlyCircle WITH center=BuildingA,
             radius=100m
9      SEQUENTIAL Launch_Video2:
10       // launch 2nd video drone ...
11     ON Video2_Ready:
12       SEQUENTIAL Area_Monitoring:
13         // trigger monitoring of facility ...
14     ON Video1_Ready:
15       PARALLEL FOR
16         flir_drone1 WITH Area1 AND
17         flir_drone2 WITH Area2 AND
18           AS flir_drone WITH SearchArea:
19             SEQUENTIAL FLIR_Search:
20               flir_drone TakeOff
21               flir_drone FlyTo WITH SearchArea
22               flir_drone Search WITH SearchArea
23     SEQUENTIAL Radiation_Detection:
24       sensor_drone TakeOff
25       sensor_drone FlyToAndPatrol WITH StartPos
26       sensor_drone DetectRadiation
27   ON Radiation_Detected BY sensor_drone WITH
         Location:
28     beacon_drone1 DployBeacon WITH beaconType=Rad,
           target=Location
29   ON Body_Detected BY flir_drone WITH Location:
30     beacon_drone2 DployBeacon WITH beaconType=Body,
           target=Location
```

Listing 2: Illustration of our mission specification language based on the example scenario in Section I. Language keywords are depicted in bold black, actors are depicted in blue, and capabilities are depicted in orange. Words without special typeset are either parameters or identifiers. Note, that capabilities realize a custom behavior that is specified separately elsewhere. For example, capability *FlyTo* performs a standard waypoint flight, while *FlyCircle* (flying along a circular trajectory) and *FlyToAndPatrol* (patrolling a target location in a pattern) realize customized behaviors.

To maintain a safe distance during take off between *video_drone1* and the two *flir_drones*, the latter ones only take off (line 14f) after the former one has reached a height of $20\,m$ (line 5). Both *flir_drones* perform their search for victims as defined in the **SEQUENTIAL** *FLIR_Search*-block (line 19). Since declared within a **FOR**-block (line 15), the enclosed block (line 19) is executed for each of the two concrete actors *flir_drone1* and *flir_drone2*, together with a certain *Area* that is provided as an argument for each of them (lines 16 and 17). Within every execution block, the actor for which the block currently executes is referred to using the alias *flir_drone* (line 18). Note that the **FOR**-block executes the statements of its enclosed block sequentially, but the blocks themselves get started in parallel, because the **FOR**-block it declared

[2]https://github.com/akio/rosgo
[3]https://github.com/aler9/goroslib

as **PARALLEL** (line 15). *Radiation_Detection* (line 23) is the third parallel *Start_Mission* block where the *sensor_drone* processes its execution blocks sequentially. The capabilities *Search* and *DetectRadiation* signal the events *Body_Detected* and *Radiation_Detected* together with the target location, when a body or a radiation has been detected respectively. Dedicated event handlers for these events (lines 27 and 29) trigger the deployment of appropriate beacons by *beacon_drones* (lines 28 and 30).

The corresponding capability *DployBeacon* is illustrated in Listing 3. Its signature (line 1) specifies that the capability is suited for a single drone with the arguments for the beacon type and its offload position. While no dedicated **REQUIREMENTS** are defined (line 2) and the beacon type is set to *Body* if no corresponding argument has been set by the caller (line 3), the **PERFORM**-block defines the process of loading a beacon of the specified type from the beacon inventory (cf. Figure 1) and deploying it at the given position (lines 5-12). Note that the *FlyTo*-statements in lines 6 and 8 refer to the capability named *FlyTo*, while line 11 makes a call to the middle-layer-API function *FlyToPosition(Position)*. If any of the involved statements throw exceptions that indicate a *critical battery level* or a *connection loss*, corresponding handlers are defined in lines 14 and 16.

```
1  CAPABILITY DployBeacon BY drone WITH Type,Position:
2     REQUIREMENTS:
3     DEFAULTS: type:Body
4     PERFORM:
5        IF type:Body
6           drone FlyTo Body_Beacon_Inventory
7        IF type:Radiation
8           drone FlyTo Radio_Beacon_Inventory
9        drone.DescentTo(3m)
10       drone.LoadBeacon()
11       drone.FlyToPosition(Position)
12       drone.Offload()
13    EXCEPTIONS:
14       ON EXCEPTION BatteryCritical:
15          // handle critical battery level ...
16       ON EXCEPTION ConnectionLost:
17          // handle a connection loss ...
```

Listing 3: Illustration of a mission capability from the mission example listing.

VI. Conclusion

In this paper, we have introduced a specification language and an execution architecture for multidrone missions aiming at domain-independence, ease of use, and expandability. Key features of our approach are mission capabilities, which compose low-level drone functionalities into mission building blocks, and a layered execution architecture, which uses Go as intermediate language. Ongoing work focuses on finalizing the execution architecture development and integration in the simulation environment AirSim [22]. Future work includes a performance study of our approach, an integration with ROS-based drones, deploying our approach in a multidrone case study (cf. https://uav.aau.at), and conducting user studies.

References

[1] H. Shakhatreh, A. H. Sawalmeh, A. Al-Fuqaha, Z. Dou, E. Almaita, I. Khalil, N. S. Othman, A. Khreishah, and M. Guizani, "Unmanned Aerial Vehicles (UAVs): A Survey on Civil Applications and Key Research Challenges," *IEEE Access*, vol. 7, pp. 48 572–48 634, 2019.

[2] J. Scherer, B. Rinner, S. Yahyanejad, S. Hayat, E. Yanmaz, T. Andre, A. Khan, V. Vukadinovic, C. Bettstetter, and H. Hellwagner, "An Autonomous Multi-UAV System for Search and Rescue," in *Proceedings of the First Workshop on Micro Aerial Vehicle Networks, Systems, and Applications for Civilian Use (DroNet)*, 2015, pp. 33–38.

[3] E. Yanmaz, S. Yahyanejad, B. Rinner, H. Hellwagner, and C. Bettstetter, "Drone networks: Communications, coordination, and sensing," *Ad Hoc Networks*, vol. 68, pp. 1–15, January 2018.

[4] J. Scherer and B. Rinner, "Multi-Robot Persistent Surveillance With Connectivity Constraints," *IEEE Access*, vol. 8, pp. 15 093–15 109, 2020.

[5] K. Dantu, B. Kate, J. Waterman, P. Bailis, and M. Welsh, "Programming Micro-Aerial Vehicle Swarms with Karma," in *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, 2011, pp. 121–134.

[6] L. Mottola, M. Moretta, K. Whitehouse, and C. Ghezzi, "Team-Level Programming of Drone Sensor Networks," in *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, 2014, pp. 177–190.

[7] D. Dedousis and V. Kalogeraki, "A Framework for Programming a Swarm of UAVs," in *Proceedings of the 11th PErvasive Technologies Related to Assistive Environments Conference*, 2018, pp. 5–12.

[8] C. Pinciroli and G. Beltrame, "Buzz: An extensible programming language for heterogeneous swarm robotics," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2016, pp. 3794–3800.

[9] M. Molina, A. Diaz-Moreno, D. Palacios, R. A. Suarez-Fernandez, J. L. Sanchez-Lopez, C. Sampedro, H. Bavle, and P. Campoy, "Specifying Complex Missions for Aerial Robotics in Dynamic Environments," in *Proceedings of the International Micro Air Vehicle Conference and Competition*, 2016, pp. 1–8.

[10] D. C. Silva, P. H. Abreu, L. P. Reis, and E. Oliveira, "Development of a flexible language for mission description for multi-robot missions," *Information Sciences*, vol. 288, pp. 27–44, December 2014.

[11] A. Rutle, J. Backer, K. Foldøy, and R. T. Bye, "CommonLang: a DSL for defining robot tasks," in *CEUR Workshop Proceedings*, vol. 2245, 2018, pp. 433–452.

[12] S. García, P. Pelliccione, C. Menghi, T. Berger, and T. Bures, "High-Level Mission Specification for Multiple Robots," in *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*, 2019, pp. 127–140.

[13] B. Schwartz, L. Nägele, A. Angerer, and B. MacDonald, "Towards a graphical language for quadrotor mission," in *Proceedings of the 5th International Workshop on Domain-Specific Languages and models for ROBotic systems*, 2014, pp. 1–4.

[14] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, "RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications," in *Proceedings of the International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, 2012, pp. 149–160.

[15] P. Doherty and J. Kvarnström, *Temporal Action Logic*. Elsevier, 2008, ch. 18.

[16] M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, P. Pillai, and J. D. Campbell, "A Language for Large Ensembles of Independently Executing Nodes," in *Proceedings of the International Conference on Logic Programming*, 2009, pp. 265–280.

[17] D. Bozhinoski, D. D. Ruscio, I. Malavolta, P. Pelliccione, and M. Tivoli, "FLYAQ: Enabling Non-expert Users to Specify and Generate Missions of Autonomous Multicopters," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 801–806.

[18] J. A. Besada, L. Bergesio, I. Campana, D. Vaquero-Melchor, J. Lopez-Araquistain, A. M. Bernardos, and J. R. Casar, "Drone Mission Definition and Implementation for Automated Infrastructure Inspection Using Airborne Sensors ," *Sensors*, vol. 18, pp. 1–29, 2018.

[19] A. Lamping, J. Ouwerkerk, N. Stockton, K. Cohen, M. Kumar, and D. W. Casbeer, "FlyMASTER: Multi-UAV Control and Supervision with ROS," in *Proceedings of the 2018 Aviation Technology, Integration, and Operations Conference*, 2018.

[20] "ArduPilot," https://ardupilot.org/planner/, accessed December 2020.

[21] "MPS - Meta Programming System," https://www.jetbrains.com/mps/, accessed December 2020.

[22] "AirSim," https://github.com/microsoft/AirSim, accessed December 2020.