EAMOS: Execution of Aerial Multidrone Mission Operations and Specifications Framework

Markus Gutmann and Bernhard Rinner

***Abstract*— Tools for specifying and executing multidrone missions that go beyond pure orchestration of waypoints are rare. We present the EAMOS framework, which introduces a simple and intuitive text-based mission specification process to execute a multidrone mission onboard different heterogeneous drones. Key benefits of EAMOS are the easy handling of sequential and parallel drone actions and their automatic synchronization. A uniform drone-interface abstracts the handling of different drone types, and specialized mission control structures enable specifying high-level missions. Our EAMOS prototype has been completely implemented in Go and successfully demonstrated in combination with the Airsim multidrone simulation environment and the PX4 flight controller as a software-in-the-loop component. Synchronization among multiple drones wrt. their sequentially and concurrently performed actions as well as the correct application of mission control structures behave as expected.**

I. Introduction

Unmanned aerial vehicles (UAVs) have proven to be a powerful tool for many use-cases in civil, governmental and military domains such as search-and-rescue, defense and surveillance, agriculture, logistics or for entertainment purposes [1]. While drone development and deployments are rapidly evolving, we observe that the availability of domain-independent, highly general, easily extendable, platform- and use-case-agnostic mission specification and execution tools is lacking behind. Besides a large number of enterprise mission planning tools that provide sophisticated graphical user interfaces and a manifold of features, the landscape of free and open-source research tools, whose mission planning goes beyond a pure orchestration of waypoints is comparatively small [2].

In this paper, we introduce the *Execution for Aerial Multidrone Operations and Specifications Framework EAMOS* (Figure 1) to close the gap of multidrone mission specifications described above. Its novelty lies in the intuitive and simple syntax for utilizing parallel drone operations, which can easily be deployed and efficiently executed onboard heterogeneous drones. Moreover, *EAMOS* combines all key aspects of a drone mission execution stack under one architecture, which supports the framework's expandability and maintenance. An *EAMOS* prototype has been completely implemented and integrated with the multidrone simulator Airsim and the PX4 flight controller.

In the following, we briefly discuss related work in drone mission specification. One example of textually specifying

Both authors are with the Institute of Networked and Embedded Systems, University of Klagenfurt, Austria (dronehub Klagenfurt: `uav.aau.at`). Email: `markus.gutmann@aau.at` and `bernhard.rinner@aau.at`

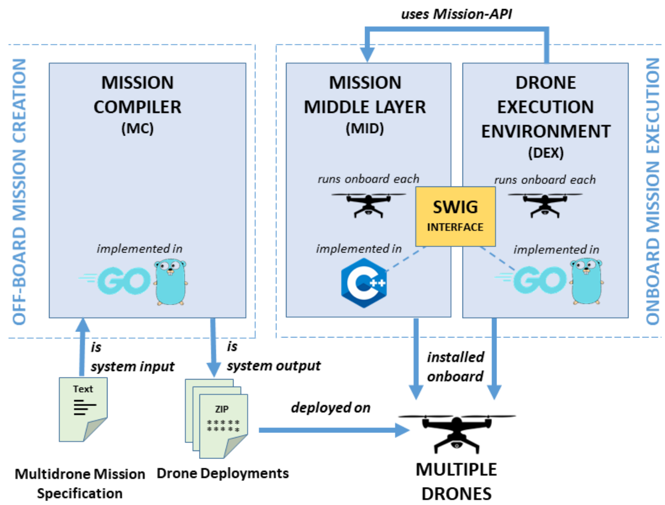

Fig. 1: Overview of the Multidrone Mission Framework *EAMOS*. Multidrone Mission Specifications are fed into the *Mission Compiler*, which is generating drone-deployments that are executed onboard by *Drone Execution Environments* utilizing the *Mission Middle Layer* to interface with drone platforms.

drone missions is the XML-based approach TML[1] [3], which is easy to comprehend but limited when it comes to expressing the control flow of a mission. A similar approach comes from Torres-González et al. [4], who propose a system for specifying cinematographic drone missions by using both graphical support and XML-based mission specification. A different approach named Papyrus by Radermacher et al. [5] specifies missions through high-level tasks that are expressed as behavior trees and get executed in a role-based layered environment based on the RobMoSys project. Alves et al. [6] propose the custom domain-specific language DRESS-ML, whose Syntax reminds on SQL. It is used to specifically define ”exceptional scenarios” that take action under special circumstances. DRESS-ML code is directly translated into the language of the target platforms.

II. Framework Architecture

A. Overview

Our *EAMOS* framework is composed of the three main components *Mission Compiler (MC)*, *Drone Execution Environment (DEX)* and *Mission Middle Layer (MID)*. The *MC* is in charge of reading and processing multidrone mission specifications (Section III) to generate individual deployment packages for the drones involved in the mission. A deployment for a particular drone contains the individual drone

[1] Task-based Mission Specification Language

mission for that drone, which is derived from the overall multidrone mission. The *MC* is a stand-alone application, which is independent of other components and runs off-board on any computer (Section IV).

The other two components *DEX* and *MID* are running onboard each drone involved in the multidrone mission. The *DEX* is in charge of initially processing and eventually executing the drone's mission, which primarily concerns monitoring dependencies across drones and executing actions onboard drones (Section V). Drone actions are both standard operations such as *TakeOff, FlyTo* and *Hover*, and special operations only available on specific drone platforms. *EAMOS* provides the *MID* that is in charge of interfacing with the particular capabilities of a drone platform.

Moreover, the *MID* provides a generic drone Application Programming Interface (API), which defines signatures of drone actions (e.g., *FlyTo*) and properties (e.g., current X-location) across different types of drones. Thus, the *EAMOS*-API provides a *TakeOff*-action, which applies to any drone involved in *EAMOS*-missions, independent of its platform. To achieve this platform independence, the *MID* requires an adapter, which provides the implementation mapping from specific drone operations to generic drone actions (Section VI).

B. Execution Model and Data Transfer

EAMOS uses the programming language Go[2] to implement its execution model, to provide data and information transfer across drones, and to handle concurrent operations onboard one drone. The execution model consists of atomic drone operations (i.e., actions) that might block until certain other actions have finished executing. This blocking and releasing mechanism builds entirely on the blocking capability of Go channels, which means that a channel blocks its current thread until it receives data from it. To implement these inter-dependencies, the execution environment in *EAMOS* uses Go channels to connect drone actions based on the derived graphs *AMT*, *MEG*, or *MDG* (Section IV) so that their predecessors block and release actions appropriately.

Go channels are also utilized to transfer data between actions. For local transfers, this is done by connecting actions with channels directly. For transfers across drones, a local end of a Go channel is connected to a ROS-topic (via ROS service calls) which spans across platforms within the ROS[3] ecosystem, that connects to a remote Go channel onboard another drone. Furthermore, Go routines provide a lightweight, overhead-free and efficient way of spawning and synchronizing concurrent local drone actions.

III. Multidrone Mission Specification

A. Overview

The multidrone mission specification contains the description of the actual multidrone mission. Similar to a computer program, the multidrone mission is composed of actions and control structures, whereas the former (e.g., *TakeOff*) can be put into named functions to be reused. The core idea is to put actions into sequential or parallel blocks, which control their execution behavior and force automatic synchronization. The simple concept is that sequential calls are executed strictly one after another, whereas parallel calls start executing at the same time. Synchronization occurs when blocks are nested. If, for instance, a parallel block B_{PAR} gets called within a sequential block B_{SEQ}, the first sequential call that follows the call to B_{PAR} waits until all calls $p \in B_{PAR}$ have finished executing correctly.

In addition to the automatic synchronization of parallel blocks, calls can be marked as asynchronous, which lets them execute in parallel to whatever is called next without letting it wait for the asynchronous execution to finish. The syntax of the multidrone mission specification is shown in Listing 1. Although not executed as a Go-program, we borrow the Go-syntax because of its simplicity and compatibility with the Mission Compiler, which is also written in Go.

```
⟨mission⟩       ::= package ⟨identifier⟩⟨drone⟩ ∗ ⟨function⟩ ∗     (1)
⟨drone⟩         ::= var ⟨drone-name⟩ ⟨drone-type⟩                  (2)
⟨funcId⟩        ::= (SEQ | PAR)⟨identifier⟩                        (3)
⟨function⟩      ::= func ⟨funcId⟩ () {⟨func-body⟩}                 (4)
⟨func-body⟩     ::= (⟨statement⟩ | ⟨func-call⟩) ∗                  (5)
⟨statement⟩     ::= ⟨action-call⟩ | ⟨condition-call⟩ | ⟨func-call⟩ (6)
⟨action-call⟩   ::= ⟨drone-name⟩.⟨capability-name⟩(⟨args⟩)         (7)
⟨func-call⟩     ::= async(⟨funcId⟩) | ⟨funcId⟩ ()                  (8)
⟨if-cond⟩       ::= If(⟨condition⟩,⟨funcId⟩,⟨funcId⟩)              (9)
⟨wait-until⟩    ::= WaitUntil(⟨condition⟩,⟨funcId⟩)                (10)
⟨wait-while⟩    ::= WaitWhile(⟨condition⟩,⟨funcId⟩)                (11)
⟨repeat-until⟩  ::= RepeatUntil(                                   (12)
                    ⟨condition⟩,⟨funcId⟩,⟨funcId⟩)
⟨repeat-while⟩  ::= RepeatWhile(                                   (13)
                    ⟨condition⟩,⟨funcId⟩,⟨funcId⟩)
⟨repeat-for⟩    ::= RepeatFor(                                     (14)
                    ⟨condition⟩,⟨funcId⟩,⟨funcId⟩,⟨int⟩)
⟨pause-until⟩   ::= PauseUntil⟨condition⟩                          (15)
⟨pause-while⟩   ::= PauseWhile⟨condition⟩                          (16)
⟨pause-for⟩     ::= PauseFor⟨int⟩                                  (17)
⟨do-until⟩      ::= DoUntil(Async | Sync)⟨condition⟩               (18)
                    ,⟨funcId⟩,⟨funcId⟩
⟨do-while⟩      ::= DoWhile(Async | Sync)⟨condition⟩               (19)
                    ,⟨funcId⟩,⟨funcId⟩
```

Listing 1: EBNF description of our multidrone mission specification. Non-terminals ⟨identifier⟩, ⟨drone⟩, ⟨drone-name⟩ and ⟨drone-type⟩ expand to strings. ⟨cond-call⟩ expands to either of rules (9)–(19).

EAMOS provides specialized *Mission Control Structures* to control the execution flow through the mission. These *Mission Control Structures* are tailored to the use-cases of dynamic multidrone missions and are available as *until-* and *while*-conditions to control mission execution more conveniently. Using while- or until-conditions determines whether conditional branches are executed if the control-condition becomes true or false, respectively. Examples of *Mission Control Structures* are illustrated in Figure 2.

[2]https://go.dev/

[3]Robot Operating System (https://www.ros.org/)

```
1 var Drone1 DroneType1
2 var Drone2 DroneType2
3 var Drone3 DroneType3
4
5 func INIT() {SEQ_perform_mission()}
6
7 func SEQ_perform_mission() {
8  PAR_arm_drones()
9  PAR_go_drones()
10  PAR_disarm_drones()}
11
12 func PAR_arm_drones() {
13  Drone1.Arm()
14  Drone2.Arm()
15  Drone3.Arm()}
16
17 func PAR_disarm_drones() {
18  Drone1.Disarm()
19  Drone2.Disarm()
20  Drone3.Disarm()}
21
22 func PAR_go_drones() {
23  SEQ_go_drone1()
24  SEQ_go_drone2()
25  SEQ_go_drone3()}
26
27 func SEQ_go_drone1() {
28  If(Drone1.Sensor1() >= 100, SEQ_go_right,
        SEQ_go_left)
29  Drone1.Land()}
30
31 func SEQ_go_drone2() {
32  Drone2.Takeoff(10)
33  PAR_drone2()}
34
35 func SEQ_go_drone3() {
36  WaitUntil(Drone1.X() >= 5 || Drone1.X() <= -5,
37  SEQ_drone3_meet_drone1)}
38
39 func SEQ_drone3_meet_drone1() {
40  Drone3.Takeoff(3)
41  async(SEQ_sense_drone3)
42  Drone3.FlyByX(Drone1.X())
43  Drone3.Land()}
44
45 func PAR_drone2() {
46  Drone2.Orbit()
47  RepeatWhile(Drone3.Y() >= 3, SEQ_picture,
48     SEQ_Drone2_Land)}
49
50 func SEQ_go_right() {
51  Drone1.Takeoff(3)
52  Drone1.FlyByX(5)}
53
54 func SEQ_go_left() {
55  Drone1.Takeoff(4)
56  Drone1.FlyByX(-5)}
57
58 func SEQ_picture() {
59  Drone2.TakePicture()
60  PauseFor(3)}
61
62 func SEQ_sense_drone3() {Drone3.Sensor2()}
63 func SEQ_drone2_land() {Drone2.Land()}
```

Listing 2: Example multidrone mission description using the Go-syntax. This mission involves three drones, organizes its actions in multiple sequential and parallel functions and uses *Mission Control Structures* such as an *If*- and *WaitUntil*-condition to control the mission flow. The mission is described in Section III-B and graphically illustrated in Figure 3.

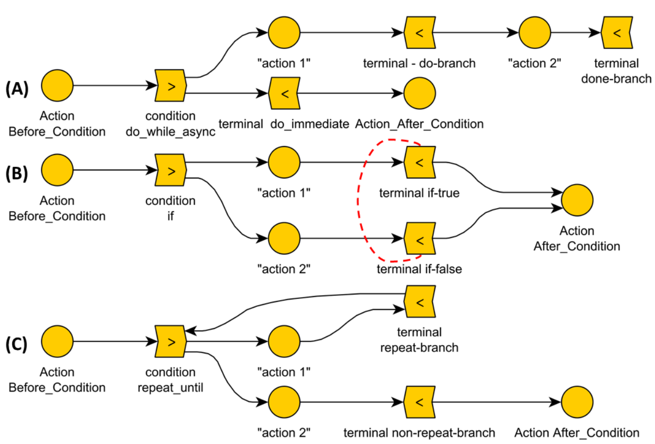

Fig. 2: Examples of *Mission Control Structures* using circles as actions and right/left angle brackets as condition nodes: (A) the asynchronous *DoWhile*-cond.; (B) the *If*-cond. The red dashed line indicates a transfer of the condition outcome so that the if-branches know which one to continue and which one to terminate. (C) the *RepeatUntil*-cond.

*1) **If**:* *If* tests a condition and either launches the *true-branch* or *false-branch*. Every branch is terminated by a *terminal node* in the execution flow.

*2) **WaitUntil, WaitWhile**:* *Wait* tests a condition and either blocks execution of the *wait-branch* until it becomes true or executes it as long as it is true.

*3) **RepeatUntil, RepeatWhile, RepeatFor**:* *Repeat* tests a condition and executes the *repeat-branch* until it becomes true (*RepeatUntil*), while it is true (*RepeatWhile*) or for a constant number of times (*RepeatFor*), before it loops back to its starting point to repeat the process. If the condition prohibits repetition, the *non-repeat-branch* gets launched instead.

*4) **PauseUntil, PauseWhile**:* *Pause* tests a condition and blocks execution of the mission until the condition becomes true or as long as it is true.

*5) **DoUntilAsync, DoWhileAsync, DoUntilSync, DoWhileSync**:* Depending on the condition outcome, *Do* launches the *do-branch* or continues with immediate execution. If the condition outcome changes, execution is interrupted, and the *done-branch* is launched. The asynchronous *Do* launches the *do-branch* asynchronously from its origin branch. The synchronous *Do*-version executes the *do-branch* and jumps to the *do-terminal* if interrupted by the condition becoming true or false, respectively. It continues with execution that was blocked before.

B. Example Mission

To illustrate the basic features of our *EAMOS Framework*, we use a simple demonstration scenario (Listing 2 and Figure 3). Here, three drones perform three basic actions in parallel (#7f.): Initially, all three drones are armed at their

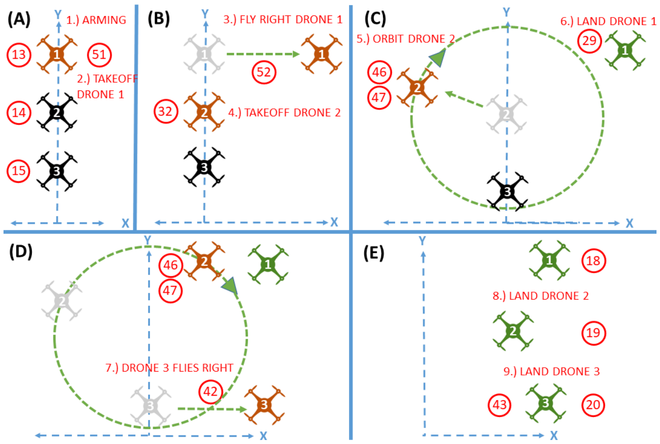

Fig. 3: Mission illustration from Listing 2 depicting three drones in the x/y plane at different time points. Red numbers represent lines in Listing 2. Colors indicate: black–armed, red–moving, grey–previous location, green–landed. (A) Drones are armed and Drone1 takes off. (B) Drone1 reads a value ≥ 100 and flies right; Drone2 takes off in parallel. (C) Drone1 lands and Drone2 starts orbiting and making pictures. (D) Drone2 keeps orbiting, Drone3 takes off and flies right, because Drone1 flew right before. (E) All drones have landed.

locations which have x-values of 0 (#12f.). Next, Drone1 performs a sensor reading, and if the sensed value is greater than 100 (#28) it takes off and either flies right or left by an x-value of 5 (#52) or by an x-value of -5 (#56), respectively. Once Drone1 has reached either of its two positions, it lands (#29). In parallel to that (cf. #22), Drone2 takes off (#32) and starts orbiting (#46) while it continuously (#47) takes pictures (#59) as long as Drone3's height is higher than 3 (#47). Drone3 waits in parallel (cf. #22) for Drone1 to either reach a location with an x-value greater than 5 or smaller than -5 (#36). If this happens (#39), Drone3 takes off (#40), performs a sensor reading (#62), flies to the x-location of Drone1 (#42), and lands (#43). If Drone3 drops below a height of 3 (#47), Drone2 stops orbiting and lands (#63). After all drones have landed, they get disarmed (#10.).

IV. Mission Compiler

EAMOS's Mission Compiler (MC) is in charge of compiling multidrone mission specifications into individual drone deployment packages that are executed onboard drone platforms. The *MC* is organized into five major components, which process the input mission in a step-wise manner. During these processing steps, every compilation stage generates intermediate outputs that serve as input for their successive stage until the final deployments are generated. Figure 4 shows the processing stages together with the outputs of the *MC*. The *MC* is completely implemented in Go.

A. Mission Parsing

In the first stage, the *MC* reads a multidrone mission specification and uses Go's source code parser to construct a complete abstract syntax tree (AST). This tree serves as a base for constructing mission objects such as drone actions, sequential or parallel execution branches, or mission control structures.

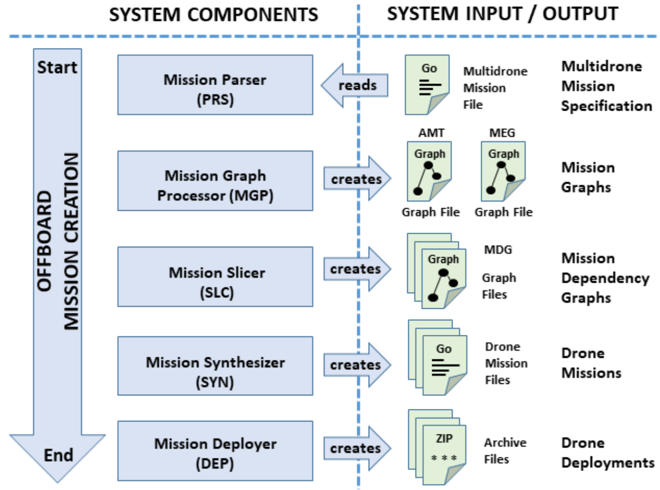

Fig. 4: System components of the *EAMOS*'s *Mission Compiler*.

B. Static and Dynamic Mission Graphs

The Mission Graph Processor receives mission objects from the *MC* and generates the *Abstract Mission Tree (AMT)* and the *Mission Execution Graph (MEG)*. The former reflects the static structure of a mission, while the latter reflects the dynamic execution flow of a mission. In the *AMT*, parent nodes reflect sequential and parallel functions, while leafs reflect drone actions.

More formally, the *AMT* is a connected, directed graph $AMT = (V_A, E_A)$ with vertex set $V_A = \{i\} \cup A \cup S \cup P \cup C \cup T$ and a partially labeled arc set $E_A = (Sources \times Targets)$, where i is the initial node, A is the set of drone actions, S is the set of sequential nodes, P is the set of parallel nodes, C is the set of condition nodes, T is the set of terminal nodes for mission condition branches, $Sources = \{i\} \cup S \cup P \cup C$ and $Targets = A \cup S \cup P \cup T$. Arcs $e \in (S \times Targets)$ are labeled with a sequence number that reflects the order in which the sequential children $t \in Targets$ are specified (and executed later). When creating an *AMT* from Listing 2, sequential blocks (e.g., #7 and #27) become sequential nodes in S, parallel blocks (e.g., #12 and #22) become parallel nodes in P, and drone actions (e.g., #13 and #51) become leafs of the *AMT*. Figure 5 shows the *AMT* of the multidrone mission from Listing 2.

In the next processing step, the *AMT* is used to create the *Mission Execution Graph (MEG)*. By imposing an order among nodes, the *MEG* reflects the mission's execution flow. Directions of arcs specify predecessors and successors of nodes implementing a simple execution model in which a node can only execute if all predecessors have finished executing, and a node triggers all of its successors once it finished its own execution.

The *MEG* is a connected, directed graph $MEG = (V_M, E_M)$ with vertex set $V_M = V_A \setminus S$ and arc set $E_M = (Sources_M \times Targets_M)$, where $Sources_M = \{i\} \cup A \cup P \cup C \cup T$ and $Targets_M = A \cup P \cup C \cup T$. In particular, since the *MEG* reflects the dynamic execution order of actions, it does not have any vertices from S anymore, because all children

Algorithm 1 MEG generation.

Input: Abstract Mission Tree AMT
Output: Mission Execution Graph MEG

for all $d \in depths$ from d_{max} to 0 **do**
    for all $s \in SEQ_d$ **do** $\triangleright$ SEQ_d: all s with depth d
        $children \leftarrow$ sorted children of s wrt. exec.order
        for $i \leftarrow 0$ to $|children|$ **do**
            $child \leftarrow children[i]$
            $leafs \leftarrow$ all leafs of sub tree w. root $child$
            for all $l \in leafs$ **do**
                $sibling \leftarrow children[i+1]$
                connect l and $sibling$ by an arc
            end for
        end for
    end for
    $parent \leftarrow$ parent of s
    link $parent$ with first child of s by an arc
    remove s from AMT
end for

of sequential vertices get recursively chained together in their corresponding order. Both the *AMT* and the *MEG* have vertex i as their root.

We convert the *AMT* to the *MEG* by "closing" all sequential vertices of the *AMT* (cp. Algorithm 1). Closing a sequential vertex s with children $c_0, c_1, \ldots, c_n$ means that all children are chained together by arcs according to their execution order, yielding new arcs $(c_0, c_1), (c_1, c_2), \ldots, (c_{n-1}, c_n)$. Finally, the parent p of s is chained to c_0 by the new arc (p, c_0), and s is eventually removed from the graph. Since the children of s can be arbitrarily complex sub trees on their own, closing must consider two points: First, the algorithm processes sequential vertices in a descending order wrt. their depths within the *AMT*, starting with those that have the highest depths. Second, when a child c_i is chained with its sibling c_{i+1} (that comes next wrt. its sequence number) and c_i is a sub tree T, the algorithm takes all leafs $l_0, l_1, \ldots, l_m$ of T to create new arcs $(l_0, c_{i+1}), (l_1, c_{i+1}), \ldots, (l_m, c_{i+1})$ to chain c_i to c_{i+1} which is chaining T to c_i respectively. Figure 6 shows the generated *MEG* of the multidrone mission from Listing 2.

C. Mission Slicing

To distribute the multidrone mission over all involved drones, the global *MEG* is used as a base to create individual *Mission Dependency Graphs (MDG)*. These graphs contain only parts of the multidrone mission relevant for a particular drone. *MDG* for drone d contains all (local) actions that are supposed to execute onboard d as well as all dependencies to other drones (i.e., external actions that are predecessors of local actions or external actions that are successors of local actions). Hence, a *MDG* is no sub-graph of a *MEG*. More formally, a *MDG* is a directed graph $MDG = (V, E)$ with vertex set $V = V_E \cup V_L$ and arc set $E = E_L \cup E_E \cup E_P$, where V_L are local actions and V_E are external actions, $E_L = (V_L \times V_L)$, $E_E = (V_E \times V_L) \cup (V_L \times V_E)$, and

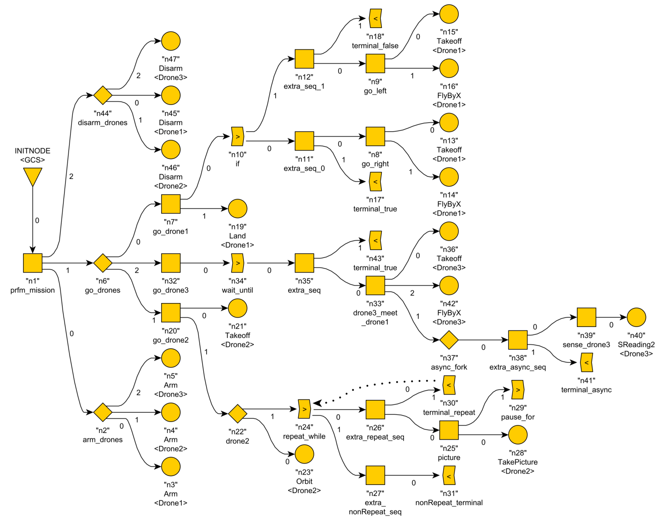

Fig. 5: *Abstract Mission Tree (AMT)* of the multidrone mission in Listing 2. Triangle: initial mission node (#7); squares: sequential nodes (e.g., #11), diamonds: parallel nodes, also called *forks* (e.g., #17); circles: action nodes (e.g., #13 or #18); right pointer: condition nodes (e.g., #36); left pointer: terminals for condition branches (not specified in Listing 2). Labels of sequential nodes, forks and condition nodes denote (node id/node type/mission identifier) and action nodes denote (node id/action name/drone name). Labels of arcs reflect execution order of sequential children. The dotted arc reflects the Repeat-Loop (#47)

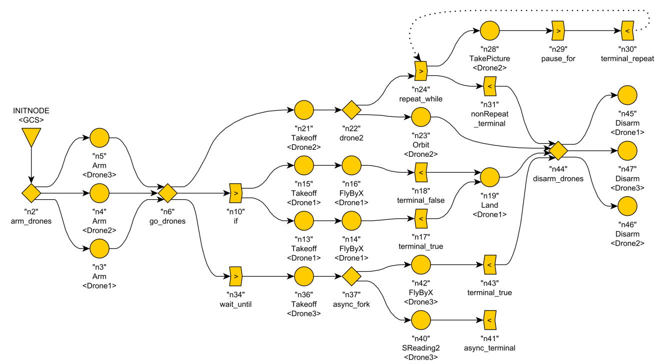

Fig. 6: *Mission Execution Graph (MEG)* of the multidrone mission in Listing 2. The *MEG* reflects execution order of the multidrone mission: A node can only execute, if all of its predecessors have finished executing. Forks trigger all successive nodes at the same time. Conditions spawn multiple branches, which are executed depending on evaluation during mission runtime. Terminals are inserted to terminate branches and to synchronize with terminals of other branches. Arc from "n30" to "n24" is a loop from #47 of Listing 2. Control vertices (forks or conditions) do not have a drone assigned to it, because these nodes were just introduced to implement the structure of the multidrone mission.

E_P are arcs that connect the terminal of a true-branch of an if-condition with its counterpart, which is the terminal of the false-branch of the same if-condition. More specifically, V_E consists of all external vertices that are immediate predecessors or immediate successors to local vertices. Since an *MDG* consists of a subset of not necessarily connected nodes of its origin *MEG* and of nodes from other *MDG*, the *MDG* is not necessarily a connected graph.

Figure 7 shows the *MDG* for Drone1 of our multidrone mission scenario, which is based on the *MEG* in Figure 6. The *MEG*-to-*MDG* conversion works as follows. For every drone d involved in the *MEG*, one *Mission Dependency Graph* MDG_d is created. For creating MDG_d, the conversion algorithm considers all vertices $v \in V_{MEG}$ that belong to drone d and sequentially performs the following three processing steps.

*1) **Forward Processing for Forks:*** Given a current vertex $v \in MEG$ that belongs to d, and one successive fork f connected by arc (v, f), vertex f is added to MDG_d. Next, all paths $path_i$ starting at f, which do only consist of forks followed until the first node other than a fork n_i for every $path_i$ is reached. Assuming that n_i is assigned to drone k, vertex f is cloned to the new vertex f_k for every encountered drone k and the new arc (v, f_k) is added to MDG_d. This process for f repeats for all successors of v.

*2) **Node Processing for all nodes other than Forks:*** Given a current vertex $v \in MEG$ other than of type fork and that belongs to d, v is added to MDG_d. Next, all predecessors $pre \in Pred_v$ and successors $suc \in Succ_v$ are connected to v by adding arcs (pre, v) and (v, suc) to the *MDG*. Here, predecessors and successors from drones other than d are considered as well, which introduces *external nodes* to the *MDG*.

*3) **Backward Processing for Forks:*** Given a current vertex $v \in MEG$ that belongs to d, and one preceding fork vertex f, connected by arc (f, v), all backward paths $path_i$ that only consist of forks are iterated backwards until the first node n_i other than a fork for every $path_i$ is encountered. During this iteration, all vertices and arcs of $path_i$ are added to MDG_d. Especially, every endpoint of path $path_i$ is linked by external or internal arcs, depending on whether or not the drone of n_i is the same as the drone of the previous vertex of $path_i$. While forks did not belong to any drone before this step, they are now assigned to drone d. This process for f repeats for all predecessors of v.

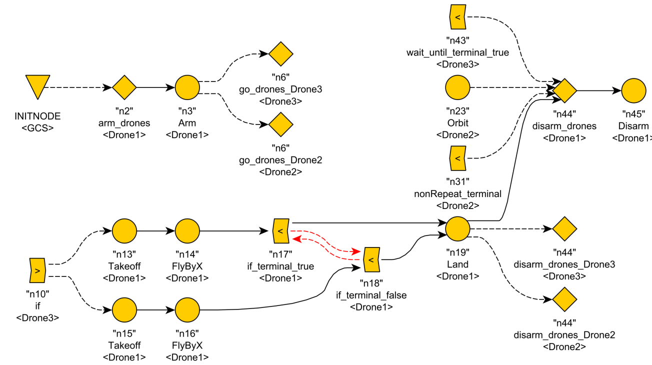

Fig. 7: Mission Dependency Graph for Drone1 of the mission of Listing 2. Solid arcs reflect internal arcs connecting local actions, dashed arcs reflect connections between internal and external nodes, and red dashed arcs reflect partner connections between the true-terminal and the false-terminal. Note that control vertices such as forks and conditions are now assigned to drones.

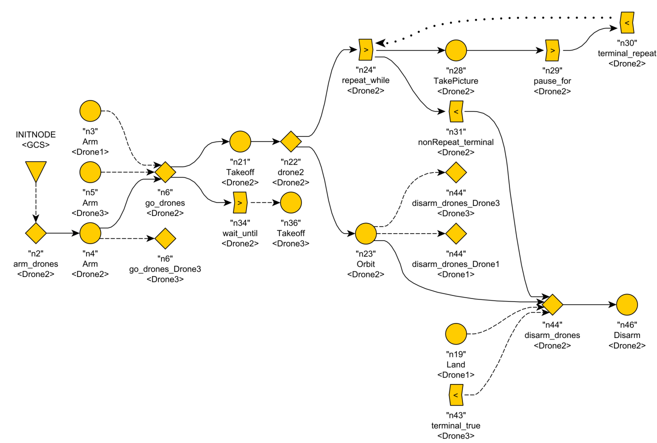

Fig. 8: Mission Dependency Graph for Drone2 of the multidrone mission of Listing 2 derived from the MEG in Figure 6. The dotted arc from n30 to n24 represents a loop that originates from a Repeat-Condition. All other features are as described in Figure 7

D. Mission Synthesizing

Mission synthesizing translates the MDG_d into a drone mission M_d for all drones d by generating executable Go-files. The M_d-file is executed onboard drone d by its *Drone Execution Environment* (Section V).

Synthesizing the M_d-file encodes all information about all internal nodes of the mission and all external nodes that have any dependencies to internal nodes. Code for internal nodes describes the associated drone actions and their internal and external dependencies. External nodes just describe their dependencies to internal nodes. Besides nodes that represent drone actions, mission control structures are also encoded into the M_d-file. The structure of the M_d file consists of a static setup-part and a dynamic runtime-part.

1) Drone Mission File Setup: For setup, node-objects (actions, forks, conditions, terminals), which correspond to the vertices of MDG_d, are declared. Every node-object-declaration defines its dependencies to preceding nodes, which correspond to the arcs of the MDG_d. Setup also links node-objects together according to their dependencies. This linking generates an executable structure in which every node knows its internal and external prerequisites to trigger the start of its local or external executions. Setup is performed once at start-up of the multidrone mission onboard drone d.

2) Drone Mission File Execution: Once the mission node structure has been fully established during setup, all nodes are initially started by what lets them listen to their incoming channels, checking whether any of their prerequisite nodes signal the finishing of their execution.

E. Mission Deployment

In a first stage of mission deployment, source files of all synthesized missions M_d together with files for the *DEX* are copied to a temporary setup-space. In a second stage, these sources are compiled by the Go-compiler and executable binaries are generated. Finally, one deployment package for each drone d is assembled, consisting of executable files of the *Adapter Space*, the *Uniform Space* and the mission file M_d.

V. Drone Execution Environment

The *Drone Execution Environment (DEX)* is a management and execution environment that runs Drone Mission

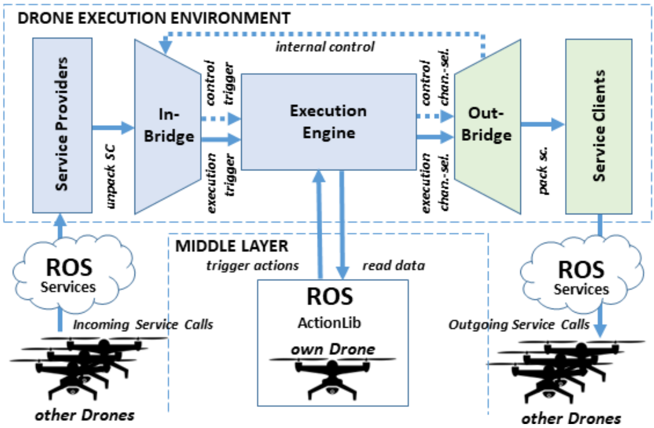

Fig. 9: Architecture of the *Drone Execution Environment*. Execution and control communication is carried out by service calls, translated into Go-channels and forwarded through bridges.

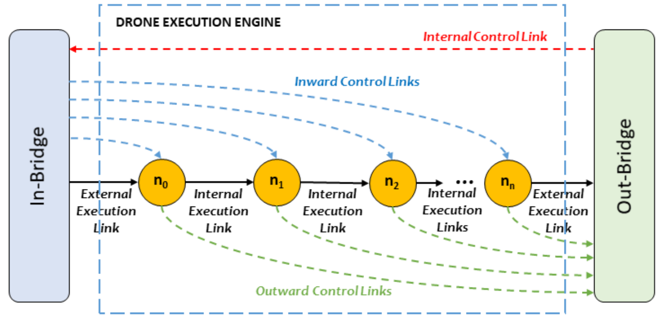

Fig. 10: Internal and external connections of actions among drones showing execution and control links.

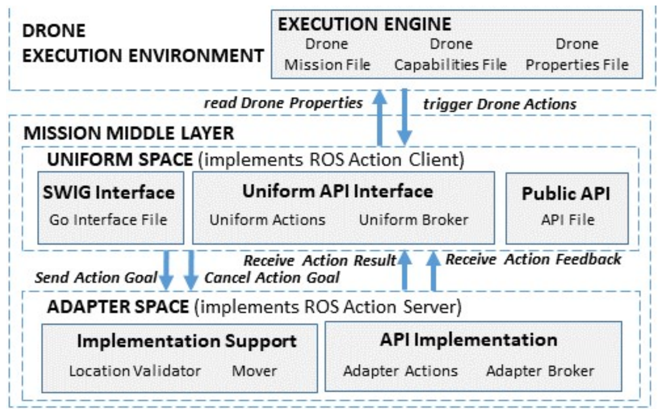

Fig. 11: Architecture of the Middle Layer, which is in charge of mapping drone capabilities and properties from the uniform API to specific drone platforms. ROS-Actions are utilized to (1) get continuous feedback about the action's progress, (2) get notified when an action goal is fulfilled and (3) be able to cancel a running action.

Files onboard drones. It is written in Go and launched onboard drones as a Go-program by the Go-runtime environment. Two main responsibilities of the *DEX* are the control and execution of mission statements and the transfer of data among nodes. Both follow internal links onboard one drone as well as external links that go beyond drone boundaries to external nodes. The environment distinguishes two types of links: (1) execution links $(n_i, n_j)_E$ connecting two nodes and enabling the execution of n_j by n_i and (2) control links $(n_i, n_j)_C$, which enable n_i to send some control messages to n_j. Control messages from n_i to n_j can, for example, trigger the execution of n_j, lock n_j for further executions, implement mission control structures that involve n_j or associate data with n_j for later processing.

Overall, the Drone Execution Environment runs ROS-nodes and facilitates the provided communication infrastructure such as ROS service calls. The architecture illustrated in Figure 9 was implemented to realize internal node executions and external node triggers. This architecture describes an Execution Engine that is in charge of executing internal nodes meaning that all nodes therein are connected by internal arcs. Two components called *InBridge* and *OutBridge* are in place for internal nodes that have external links attached. If an internal node n has an external link (execution or control) as a prerequisite, the internal node is attached to the *InBridge* by an external link $(InBridge, n)$. On the other hand, if n triggers an external node by an external execution link, n is attached to the *OutBridge* by an external outward execution link $(n, OutBridge)$. Since any node is supposed to be able to send control information to any other node, the *InBridge* connects every internal node within the Execution Engine with an inward control link, and every internal node is connected to the *OutBridge* by an outward control-link.

Communication over links is carried out by ROS service calls, which are provided by the *Drone Execution Service Provider* and called by the *Drone Execution Service Client*. An internal node n_i of drone d_i, triggering an external node of drone d_j, forwards data from n_i to the local *OutBridge* of drone d_i, packs the data into a service call that is received by the remote service provider of drone d_j, where data is read by the *InBridge* and forwarded through the external link of the corresponding internal node n_j. To send a control message from one node to another internally, an internal control link connects the local *OutBridge* with the local *InBridge* so that communication goes from $node_{src}$ to $OutBridge$ to $InBridge$ to $node_{rcv}$. This mechanism takes advantage of the *InBridge* being connected to any internal node. Figure 10 illustrates this communication scheme.

VI. Middle Layer

Since our framework supports heterogeneous fleets of drones, we define a general API of drone capabilities and properties, which can be uniformly applied to every drone. Our *Middle Layer (ML)* translates uniform API-calls downwards to specific ones and generalizes specific replies upwards to comply with the uniform API (Figure 11. The *ML* provides a *Uniform Space* and an *Adapter Space*. The first implements the uniform drone-API and is supposed to be used by drone mission files to interact with drones. The latter implements uniform API functionality targeting a specific drone platform.

A drone type is compatible with *EAMOS*, if a corresponding platform-adapter is available in the *Adapter Space* of the *ML*. Both *Uniform-* and *Adapter Space* are organized in *Actions* and *Brokers*. *Actions* implement standard drone actions such as *TakeOff* as well as more specific ones such as *Orbit*, if a platform supports them. *Brokers* deliver data from the drone to upper layers in the execution stack such as drone position or sensor readings. Uniform *Brokers* are publicly available for anyone to request drone properties. Furthermore, the *Uniform Space* provides utility implementations such as a *LocationValidator* to monitor whether a drone reached a target location, or the *Mover*, which actually moves a drone to a target location. Drone Mission files, which contain the mission-part for one individual drone, are entirely synthesized using *Go*. The *Uniform Space* of the *ML* with its uniform API is entirely implemented in *C++*. Hence, a not so trivial language-barrier exists because drone mission files and the *ML* need to work closely together. To overcome this barrier, we use the programming language interface framework *SWIG*[4] to make the Go-layers and the C++-layers work together. The *Uniform Space* provides a C++-file declaring the uniform drone API, and a SWIG-interface file mapping Go-code to C++-code and vice versa. *SWIG* converts the public C++ uniform API into Go so that drone missions can access the capabilities and properties of drones originally provided in C++, by using Go.

VII. Evaluation

For our evaluation, we set up a full ROS Noetic environment on an Ubuntu 20.04 installation that runs the MAVLINK ROS-package mavros[5], the Pixhawk flight controller PX4[6], the monitoring and interfacing tool QGroundControlStation[7] and Microsoft's Unreal-based simulation engine Airsim[8] (Figure 12). This environment enables a software-in-the-loop simulation of realistic multidrone systems. Since many real drone platforms use the same software modules (except Airsim), we can run *EAMOS* drone deployment packages without loss of generality on the corresponding *DEX* and *MID* (separate instances for every drone).

Our ongoing experiments with *EAMOS* involve different simple missions with up to four simulated drones, all showing the drones performing the expected synchronized maneuvers in accordance to their multidrone mission. Once all supporting systems are up and running, modifying a mission requires first recreating and then redeploying the multidrone mission, followed by restarting the mission onboard the simulated drone, which all happens with a few clicks. Using the Go-profiler pprof[9], we assessed *EAMOS's* **space-** and **runtime-** performance during *creating* and *starting* a mission. Creating the mission from Listing 2 (34 nodes) required 4368.78 kB and took 50 ms while starting the mission for Drone1 (22 nodes) required 3749.50 kB taking 916.63 ms. We further tested the scalability by creating a stress-test mission with more than 5000 nodes and 20 drones which

[4]http://swig.org/
[5]https://github.com/mavlink/mavros
[6]https://px4.io/
[7]http://qgroundcontrol.com/
[8]https://github.com/microsoft/AirSim
[9]https://github.com/google/pprof

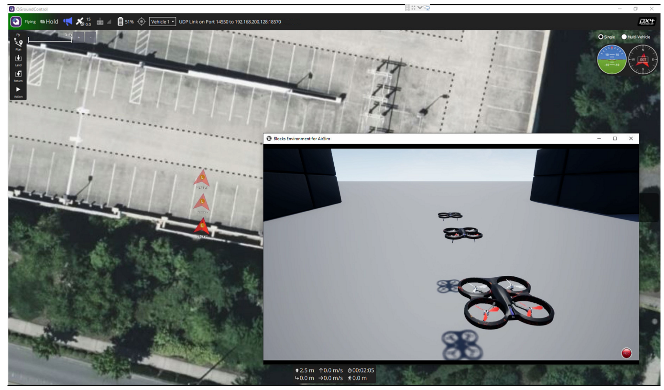

Fig. 12: Our simulation environment showing three simulated drones in Airsim (foreground) that all have their own software-in-the-loop deployment consisting of PX4, mavros and the *EAMOS*-deployment, which are all used by the QGroundControlStation (background) to visualize vehicle telemetry and indicate vehicle locations on the map.

required 23979 kB and took 2.34 s. Starting the mission for a drone with 634 nodes required 4257.45 kB taking 733.64 ms.

VIII. Conclusion

The results of our experiments suggest *EAMOS* to be a promising approach for utilizing multiple drones through continuously specifying and executing multidrone missions for them. The lightweight framework runs upon an existing ROS infrastructure and provides the complete software stack from mission specification to mission execution. It compiles arbitrarily complex but easy-to-read and easy-to-write multidrone missions and deploys them to heterogeneous drones while providing a uniform platform API. One challenge for the coming steps is to provide mechanisms that assist in formulating high-level missions with basic drone actions such as aggregating actions. We will extend our laboratory setup to replace simulated drones by real quad-copters in future work.

References

[1] B. Rinner, C. Bettstetter, H. Hellwagner, and S. Weiss, "Multidrone systems: More than the sum of the parts," *IEEE Computer*, vol. 54, no. 5, pp. 34–43, 2021.

[2] M. Gutmann and B. Rinner, "Mission specification and execution of multidrone systems," in *Proc. Design, Automation and Test in Europe Conference (DATE)*, virtual, 2021, pp. 1–6.

[3] 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 *Proc. International Micro Air Vehicle Conference and Competition*, Beijing, China, 2016, pp. 1–8.

[4] A. Torres-González, A. Alcántara, V. Sampaio, J. Capitán, B. J. N. Guerreiro, R. Cunha, and A. Ollero, "Distributed mission execution for aerial cinematography with multiple drones," in *Proc. Workshop on Signal Processing Computer vision and Deep Learning for Autonomous Systems*, A Coruña, Spain, 2019.

[5] A. Radermacher, M. Morelli, M. Hussein, and R. Nouacer, "Designing drone systems with papyrus for robotics," in *Proc. Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools*. Budapest, Hungary: ACM, 2021, p. 29–35.

[6] L. Alves, J. D. Pereira, N. Aragão, M. Chagas, and P. H. Maia, "DRESS-ML: A domain-specific language for modelling exceptional scenarios and self-adaptive behaviours for drone-based applications," in *Proc. IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, 2022, pp. 56–66.