IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 18 October 2023, accepted 31 October 2023, date of publication 6 November 2023, date of current version 13 November 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3330652

RESEARCH ARTICLE

Multidrone Mission Execution With EAMOS: From Text to Mission

MARKUS GUTMANN AND BERNHARD RINNER, (Senior Member, IEEE)
Institute of Networked and Embedded Systems, University of Klagenfurt, 9020 Klagenfurt, Austria
Corresponding author: Bernhard Rinner (bernhard.rinner@aau.at)

ABSTRACT Existing software tools for specifying and executing multidrone missions are limited to route planning or tightly coupled to specific drone hardware. We introduce EAMOS (Execution of Aerial Multidrone Missions and Operations Specification Framework), which allows us to specify missions intuitively, text-based, and provides a mission compiler, a mission middle layer, and a distributed drone execution environment. The middle layer wraps the control of individual drone-specific capabilities, such as launch, fly to position, or perform a maneuver, into a public API that transparently utilizes the capabilities of numerous drone platforms. We exploit the Go programming language to implement critical components of the framework and provide an interface for ROS-based drone platforms. EAMOS automates the mission execution on real, virtual, and even hybrid robotic setups involving real and virtual drones. We demonstrate the successful deployment of EAMOS with four missions executed on Pixhawk/PX4-equipped quadcopters and virtual drones simulated with Airsim. We assess the performance of our proposed approach by analyzing the number of nodes and arcs of the mission graphs, which are an essential artifact of our mission compilation, the utilization of ROS service calls during mission execution, and the duration of compilation, deployment, and mission execution. Overall, our experiments showed that our drones correctly behaved during mission execution as expected and specified by their mission, the generated mission artifacts were efficiently manageable, and processing times allowed for a fluent workflow.

INDEX TERMS Multi-robot missions, software framework, mission execution, drones, ROS, Airsim.

I. INTRODUCTION

Over the past years, unmanned aerial vehicles (UAVs) or commonly called drones have proven to be a capable tool in diverse civil domains [1] such as agriculture [2], construction [3], inspection [4], meteorology [5], surveillance [6], photography [7], cinematography [8], consumer entertainment [9], search and rescue [10] and even space exploration [11]. Applications of drones are accordingly manifold and range from exploratory tasks such as visual image mosaicking [12] or simultaneous environment covering tasks [13] to more interactive ones such as fruit harvesting [14] or parcel delivery [15], to name just a few.

As a consequence of the comprehensiveness of this field, a large variety of unmanned aerial systems (UAS), encompassing different drone platforms and diverse software tools, already exists. Many approaches are tailored for end users with medium- to expert-level experience. While a single human operator can typically handle single drone flights, teams of drones require the support of automation because it is usually not realistic to have one human operator for each drone involved in a mission.

A. CHALLENGES OF MULTIDRONE SYSTEMS

What a drone mission is depends on the concrete use case and the application domain of a UAS. While battery utilization and onboard computer performance are typical challenges for multidrone systems from a platform perspective, uncertain weather conditions and dynamic environments are challenging to manage from a drone mission perspective (cf. [16]). A user study that involved drone users from an emergency response service (cf., [17]) revealed that having appropriate control and monitoring devices that are usable in harsh outdoor conditions is an essential challenge of

The associate editor coordinating the review of this manuscript and approving it for publication was Lei Shu.

© 2023 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

multidrone systems, together with the ability to efficiently handle multiple video feeds that are cast to multiple screens simultaneously. Furthermore, a strong support of the end user's situational awareness of what each drone is doing and how the embedding environment affects the drones was identified to be a critical challenge. Study participants mentioned that it is desirable to keep the autonomy level of missions flexible, which means that end users should be able to switch between manual and autonomous control if necessary.

As described in [18], it is of paramount relevance for a multidrone system, whether the communication network is based on a centralized or decentralized architecture. While the former makes the survival of the whole system dependent on a single node, such as the ground control station, the latter is much more resilient against the failure of individual nodes.

B. MOTIVATION FOR THIS WORK

To tackle theses challenges and limitations, an open and extendable mission specification and execution system that can adapt to different use cases and applications is needed. While building blocks of drone missions must be as generic as possible, the specification of specific mission scenarios in custom contexts should be possible at the same time. In our opinion, high-level (imperative) programming languages such as C/C++, Java, or Python exactly provide this synergy of putting together simple and generic blocks to form complex and specific mission scenarios. Although it is popular to use such languages (and their surrounding programming environments) to specify multidrone missions, it has the price of requiring skilled software developers. Thus, the overarching challenge is to develop a text-based multidrone mission specification approach with a flat learning curve for a target user group of domain experts with little to no programming skills. We further see a simple-to-create and easy-to-comprehend specification of multidrone missions as key to unleashing the full potential of a UAS and its drones.

C. CONTRIBUTIONS OF THIS WORK

We introduce the EAMOS Framework (**E**xecution of **A**erial **M**ultidrone Missions and **O**perations **S**pecification), which aims to provide both a user-friendly tool for specifying multidrone missions and a decentralized platform for executing those missions (see Figure 1). EAMOS supports any number of real and simulated drones and even supports hybrid setups involving both types of drones. We placed our execution stack on top of the middle layer framework ROS (Robot Operating System[1]), which is today's de-facto standard for robotic application developments. This allows us to use a large and growing number of supported hardware devices and utilize a rich pool of software tools to interface and work with diverse drone platforms. While the application of ROS usually requires experienced experts, mission specifications in EAMOS entirely hide technical aspects from the end

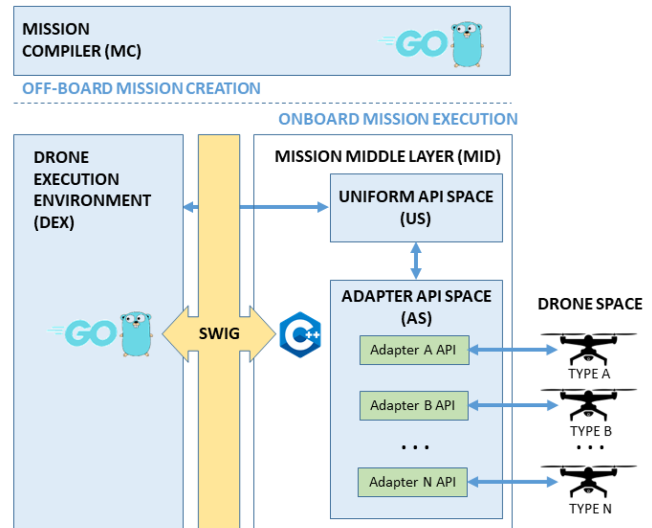

FIGURE 1. Overview of the Multidrone Mission Framework EAMOS. The Mission Compiler (MC) generates missions that are deployed onboard drone platforms on top of the Drone Execution Environment (DEX) and the Mission Middle Layer (MID). DEX executes the missions and interacts with the drone platforms via MID's Adapter APIs (blue arrows). SWIG provides a bridge between Go and C++.

user by providing a domain-specific mission language that entirely focuses on the objective of the multidrone mission rather than including code that is salient but irrelevant and introducing much syntactic overhead to the mission. The key contributions of this work can be summarized as follows.

1) **Mission Specification**: EAMOS provides a domain-specific approach for describing missions that use the syntax of the modern programming language Go and introduces mission control structures, which allows to specify missions that can deal with changing and uncertain contexts.
2) **Mission Compiler**: EAMOS compiles multidrone mission specifications into several mission graph structures and further into an intermediate mission representation, eventually compiled into executable Go programs.
3) **Mission Execution**: EAMOS provides its fully distributed mission execution environment, which is entirely written in Go to provide a high degree of compatibility with the mission specification and compilation.
4) **Decentralized Architecture**: The user easily specifies multidrone missions within a single file, while EAMOS distributes the mission only to those drones involved in it, avoiding a centralized entity for controlling the mission execution.
5) **Explicit Concurrency**: The author of multidrone missions in EAMOS explicitly states which actions are meant to be run sequentially and which should be executed in parallel. EAMOS then takes care of the scheduling and synchronization of all actions.
6) **Middle Layer**: EAMOS provides its own middle layer tier that wraps the control of individual

[1] https://www.ros.org/

drone-specific capabilities, such as taking off, flying to a position, or applying onboard sensors, into a uniform and public mission API (Application Programming Interface), which covers the capabilities of numerous heterogeneous robotic platforms. Moreover, the Middle Layer can be extended to enrich the API and support more drone platforms.

Our approach puts the user in charge of writing a meaningful and correct mission and handling critical situations. Checking the correctness and feasibility of missions is beyond the scope of this work. Since the EAMOS framework is designed generically and openly, such checks can be integrated as a preprocessing step, which would improve the usability of EAMOS even further.

While our previous work [19], [20] provided an introduction of the mission specification, an overview of the framework, and an initial demonstration with the Airsim[2] simulation environment, this paper comprehensively describes the complete EAMOS framework. It demonstrates its successful usage in single and multidrone missions on real and hybrid platforms. In particular, we present details on the mission compilation, the drone execution environment, the mission execution model and the interface to ROS-based drone platforms, and the experimental evaluation using four different experiments. To the best of our knowledge, EAMOS is the first multidrone mission specification and execution framework that facilitates an intuitive, text-based approach for describing multidrone missions that support explicit concurrency with automatic synchronization, advanced control structures, and transparent mission execution in a simulation environment, on real ROS-based drone platforms, and hybrid settings.

D. OUTLINE

The remainder of the paper is organized as follows. Section II provides an overview of the relevant related work. Section III briefly discusses the methods applied during the life cycle of a multidrone mission. Section IV presents the overall architecture of EAMOS with its major components. Section V details the essential components of EAMOS' Mission Compiler, which processes a textual mission into an executable drone deliverable. Section VI provides a detailed insight into EAMOS' Execution Environment, which ultimately executes the drone mission deliverable onboard drones. Sections VII and VIII report on the experimental setup and the results of four experiments. Section IX compares the results with related approaches. Section X concludes the paper with a summary and discusses future steps of EAMOS.

II. RELATED WORK

Our review of practically applied multidrone mission approaches revealed that missions are usually defined using high-level programming languages like C/C++ or Python. The expressiveness of those languages enables mission authors to easily define highly custom and use-case-oriented missions for real-world situations, as long as the mission author is an expert in using the specific language (i.e., is a software developer). Moreover, such mission programs tend to provide many verbose code fragments that require the mission author to write much code to achieve minor functionalities. This introduced syntactic overhead makes the encoding of a mission more dominant than its actual use case. In short, such missions are unusable by non-experts. Examples are the C++ or Python APIs for the simulation tool Airsim[2], which are already tailored to their multidrone context but still require a programmer if the mission specified goes beyond pure take-off and landing actions.

A. INVESTIGATED APPROACHES

To avoid requiring a software developer to specify robot missions in general or drone missions in particular, several new languages and dialects were already proposed as part of text-based mission specification approaches. Most of those introduce dialects of the popular data description and encoding markup language XML (Extendable Markup Language). While some related work introduced entirely custom domain-specific languages, many existing approaches use graphical maps, which come with a graphical user interface (GUI) to specify robot missions. While GUIs support the user experience and graphical maps are highly appropriate for specifying drone missions, those approaches usually hide their internal logic responsible for compiling and processing missions, so it is often unclear what models and processes are in place "under the hood". Therefore, extending the mission specification capabilities becomes difficult or even impossible. To analyze the state of the art, we focused on the qualitative aspects presented in Table 1. The investigated approaches are summarized in Table 2.

1) **"TML" (Task-based Mission specification Language)**: This approach by Molina et al. [21] expresses missions by using the concept of tasks, which aggregate atomic drone capabilities called actions, to context-related meaningful units of execution. Furthermore, the concept of skills is used for intrinsic drone capabilities. The equally named concept supports conditions that control the execution flow. All concepts in TML are available as XML tags and are used to specify static mission descriptions with XML-based markup syntax. The execution of TML missions is performed by the existing Aerostack[3] framework for which the authors wrote a specific TML interpreter. However, the XML base syntax degrades readability, making longer mission specifications verbose and hard to handle. Though multi-vehicle support seems supported, an explicit synchronization among agents or their actions seems unsupported. Conditions are supported in terms of an

[2]https://microsoft.github.io/AirSim/

[3]https://github.com/aerostack/install/wiki

TABLE 1. Qualitative aspect used for describing mission specification approaches.

Aspect	Description
Category	Approaches are either text-based defining a domain-specific language, or GUI-based, which provide the user a graphical map to specify missions.
Mission Syntax	Reflects the syntax used for specifying missions. This syntax criterion is independent of any other syntax or language used within an approach (e.g., to program components such as a compiler). Moreover, the mission syntax does not necessarily imply any processing (e.g., compilation) of the mission, because most approaches just "borrow" the used syntax from another context. For instance, XML mission-dialects are frequently used to describe dynamic mission executions, while standard XML rather describes static data hierarchies without any order among its elements.
Mission Processing	Reflects how the mission text is further processed. Missions are **interpreted** if their text is fed into a program interpreter and executed as it is (e.g., Python). Missions are **translated** if they are first translated into another form or syntax, before being further processed.
Intermediate Form	If a mission is translated, this aspect reflects the form the mission is translated into. In most cases, it is an executable program, such as a C/C++, C#, or Java program, but also custom forms such as taskplans or state machines are intermediate mission representations used in related approaches.
Execution Engine	Reflects whether the system that eventually executes the missions is also within the scope of the work of the approach.
Decentral Architecture	Reflects whether the proposed system relies on a centralized or decentralized architecture for controlling the mission execution.
Multiagent	Reflects whether missions can involve more than one drone, without necessarily supporting explicit synchronization (see also explicit synchronization).
Explicit Synchronization	Reflects whether the user is able to explicitly state how multiple drones synchronize their actions. This means in particular, that the user can explicitly state, which actions execute sequentially and which execute in parallel. Thereby, the user easily states, which actions wait for the completion of other actions, and EAMOS takes care of the necessary synchronization in the background.
Domain Independence	Reflects whether the approach is suitable for arbitrary application domains and contexts, rather than being bound to just a single applications domain.
Control Structures	Reflects whether the approach provides means to control the execution flow during the mission (e.g., if-then-else condition, do-while, etc.).
Experimental Evaluation	Reflects whether the approach was evaluated with a simulation environment or tested on real robotic platforms.

TABLE 2. Summary of related mission specification approaches and comparison with our EAMOS framework. The qualitative aspects used to characterize the approaches are detailed in Table 1. Since this work focuses on purely text-based approaches, the discussion of internal (hidden) mission syntax, mission processing, and intermediate forms of GUI-based approaches is omitted.

Superscripts: [a]The authors wrote an interpreter for the Aerostack framework (https://github.com/aerostack/install/wiki). [b]The authors describe potential translations into Petri Nets, Goal and task plans, and behaviors. Though, it is not clear, which representation was the main choice. [c]The authors describe CoolBOT as the used system framework and T-REX and MOOS as candidates for execution engines. However, it is not clear which system was used. [d]n/a means that the information was either unavailable or not clearly described in the literature. [e]The Task Description Language is created by / inputted into a GUI planner, which sends the mission directly to the AUV. [f]Although the literature mentions group tasks, it is unclear whether/how the approach supports multiple vehicles. [g]No particular form is supported. The approach describes model-generators for arbitrary target representations. [h]Taskplans are fed into the approach's Planning Module, which is in charge of mission execution.

#	Name / Approach	Year	Category	Mission Syntax	Mission Processing	Intermediate Syntax / Form	Execution Engine	Decentral Architecture	Multiagent	Explicit Synchronization	Domain Independence	Control Structures	Experimental Evaluation
1	"TML" [21], [22]	2016	Text	XML	Interpreted[a]	-	Yes	No	Yes	No	Yes	Yes	Real
2	"MDL" [23]–[25]	2014	Text	XML	Translated	C#	Yes	No	Yes	No	Yes	Yes	No
3	Fernández-Perdomo et al. [26]	2009	Text	XML	Translated	n/a[b]	other[c]	n/a[d]	No	No	No	Yes	n/a
4	"AVCL" [27], [28]	2005	Text	XML	Translated	Java	No	No	No	No	No	No	Sim.
5	Bagnitckii et al. [29]	2011	Text	custom	n/a[e]	n/a	No	No	*[f]	No	No	No	No
6	"DRESS-ML" [30]	2022	Text	GIVEN-WHEN-THEN	Translated	custom[g]	No	n/a	No	No	Yes	Yes	Sim.
7	"Director Tools" [31], [32]	2020	Text / GUI	XML	Translated	Taskplans[h]	No	No	Yes	No	No	No	Real
8	Paula et al. [33]	2020	GUI	-	-	-	Yes	No	Yes	No	Yes	No	Real
9	"FLYAQ" [34]–[37]	2015	GUI	-	-	-	Yes	No	Yes	No	Yes	No	Real
10	Besada et al. [38]	2019	GUI	-	-	-	Yes	No	Yes	No	Yes	No	No
11	"FlyMASTER" [39]	2018	GUI	-	-	-	Yes	Yes	Yes	No	Yes	No	Real
12	"EAMOS"	2023	Text	Golang	Translated	Golang	Yes	Yes	Yes	Yes	Yes	Yes	Real

event handling functionality but without supporting arbitrary conditions or control structures.

2) **"MDL" (Mission Description Language)**: Silva et al. introduced a similar markup-based approach which is based on four languages, namely Scenario Description Language, Team Description Language [25], Disturbance Description Language [24], and Mission Description Language (MDL) [23]. The authors developed this framework to provide a rich set of vocabularies and verbosely describe a multidrone mission's different aspects, such as vehicular characteristics, geographical sites, or the compositions of so-called drone teams. However, it exhibits the same drawbacks even if the approach offers a higher expressiveness compared to the TML approach presented above due to a richer set of predefined vocabularies.
3) **Approach by Fernández-Perdomo et al.**: This work [26] aims to simplify mission specifications for unmanned underwater vehicles (AUV). The authors define a mission to consist of several mission plans: a Logging Plan, a Navigation Plan, a Communication Plan, a Measurement Plan, and a Supervision Plan. While these plans are supposed to be exchangeable, each takes care of a particular aspect of the overall mission. This approach does not support multiple vehicles and is tightly coupled to its application domain of AUVs. Conditions seem to be supported in a rather inflexible manner by binding them directly to tasks.
4) **"AVCL" (Autonomous Vehicle Command Language)**: Davis et al. [27] propose a mission specification approach for AUVs, which is based on ontologies for tasking, communications, and results wrt. an AUV mission. Their approach defines a task-based specification comprising a catalog of AUV primitives for controlling the vehicle (e.g., hovering, loitering, moving, or waiting). Furthermore, missions are declarative goal-based specifications, which can be represented and processed by finite state machines (FSM). The authors created the XML-based DSL named AVCL, which is used to notate a declarative FSM that expresses an AUV mission. Moreover, a GUI named Autonomous Underwater Vehicle Workbench [28] was developed, enabling the end-user to edit the task and goal-based levels of a mission. Even the term "Rendezvous" is mentioned wrt. AUVs, the mission specification language does not seem to support multiple vehicles. The application domain is restricted since this approach addresses AUVs. Moreover, the language does not seem to support any control structures.
5) **Approach by Bagnitckii et al.**: Bagnitckii et al. [29] introduce a domain-specific task description language for specifying vehicle missions that allow the use of navigational parameters, enable AUV group configurations, and support the description of task groups for AUVs. To favor readability and usability, the authors avoided using XML syntax. A multi-vehicle mission plan can be represented as a finite state machine providing a designated start and end configuration. The presented DSL follows a hierarchical structure and declares high-level mission elements such as mission events, mission goals, forbidden areas, and time limits. By avoiding the verbose syntax of XML, the approach provides a custom and easy-to-read syntax for the mission specification. The approach comes with a feature-rich GUI for end-users. However, since targeting AUVs, the approach is restricted to the corresponding application domain. It is unclear whether the approach supports multiple vehicles, and no control structures are described.
6) **"DRESS-ML"**: The custom domain-specific language DRESS-ML, which was developed by Alves et al. [30], focuses on the description of exceptional scenarios for drones and the specification of how to handle them such as hardware component failures, detection of suspicious events (such as smoke), or drone emergencies (such as a loss of height or GPS-signal). Such situations are described with a SQL-similar syntax using the clauses "Given", "When" and "Then" (cf. "Select", "From" and "Where" of SQL). Although the described exceptional scenarios are mission fragments rather than complete drone missions, they are passed through a pipeline that encodes their source code by a so-called "ModelToText-generator" and then translates it for different target drone platforms. DRESS-ML is not a complete mission specification approach because it just covers exceptional scenarios that might occur during a UAV mission. Compared to all other investigated approaches, its syntax is tailored to its mentioned scope. However, its mission snippets offer good readability, and its integration into existing platforms seems lightweight.
7) **"Director Tools"**: Montes-Romero et al. [31], [32] propose an approach to facilitate the capturing of cinematographic shots and scenes with multiple drones. They also claim that approaches to utilize multiple drones already exist, but the expertise required to apply them exceeds those of a pure domain user. In their approach, which they simply refer to as a set of "Director Tools", they introduce a Mission Controller and a graphical user interface to assemble multidrone cinematographic shootings, which is first output as standard XML markup and then fed into a mission planner creating an executable schedule of drone actions from it. This approach is tailored to photographic and cinematographic productions that utilize multiple drones and is thus heavily domain-dependent. Moreover, the used XML-dialect makes missions hard to handle and more dependent on the GUI tools.
8) **Approach by Paula et al.**: To support the monitoring and controlling of multiple drones, Paula et al. [33] propose a GUI-based approach that covers aspects

such as drone telemetry, connected sensors, geographic coordinate-based drone control, mission execution, drone collaboration, and event logging. Missions are configured through a web interface using actions such as taking off, waiting, and landing. Even if the approach introduces a user-oriented and modern GUI, the number of actions for missions seems limited. Thus, the approach seems to be very restricted in its applicability. Moreover, the suggested system architecture describes an essential ground station for running the multidrone missions, making it a centralized approach.

9) **"FLYAQ"**: the authors of this work [34], [35], [36], [37] propose a family of three (main) DSLs, forming an approach that claims to be technology-independent, analyzable, executable and testable via simulations, extensible wrt. new application areas, and closer to the problem domain than previous approaches. The Monitoring Mission Language (MML) addresses the end-user, who is assumed to be a domain expert. The Robot Language comprises the capabilities and properties of a particular robot platform and covers to support all robot types (i.e., not necessarily all particular robots). The Behavioral Language defines atomic movements of all supported robots that can be used through the MML, such as Start, HeadTo, or Circle. MML specifications are translated into the intermediate text-based language QBL, which can then be inputted into the framework's mission execution engine. FLYAQ provides a user-friendly GUI for specifying multidrone missions, supports numerous heterogeneous platforms, and even provides a reconfiguration engine to increase the resilience of its missions. However, FLYAQ does not seem to support control conditions, nor does it allow to explicitly synchronize multiple drones, while utilizing multiple drones is supported. Moreover, looking at the FLYAQ system architecture, the approach seems to be fully centralized.

10) **Approach by Besada et al.**: This highly end-user-oriented and map-based approach from Besada at al. [38] proposes a comprehensive environment for managing drone fleets, resources, and missions. The architecture comprises two main microservices for monitoring and defining a multidrone mission and several smaller microservices for managing resources, environmental data (e.g., weather), drone sensor data (e.g., video), and user data. Moreover, the centralized approach provides four databases to store all kinds of data generated or transmitted during missions. The authors developed a multi-modal GUI (for desktops, mobile devices, and video walls), where sensors and actuators of the drones are selected, and the location, radius of the operation zone, and landing point are set. Once defined, the GUI uses the MAVLink protocol to transmit a generated mission plan to the drones. Even if this approach comes with a user-friendly GUI, it is basically a pure waypoint orchestrator. In particular, it does not support explicit synchronization among drones or provide control structures. Moreover, this approach does not seem to be tested, neither in simulation nor with real drones.

11) **"FlyMASTER"**: This work by Lamping et al. [39] covers the whole stack for managing multidrone flights. FlyMaster is developed as a ROS package built upon ROS, MAVLink, and MAVROS and uses an onboard computer (OBC) attached to the drone's flight controllers. It further uses a ground control station (GCS), which communicates with the ROS-based companion computers through the ROS packages ROSbridge and ROSlibJS. The most significant features of FlyMaster are to support resource-intensive computations, the freedom to choose where to execute ROS nodes (on the GCS or the OBC), to support telemetry streaming to the GCS, as well as relaying of commands from the GCS to the drones. The GCS is a user-friendly GUI, which serves to edit and launch multidrone missions. Once a mission is launched, the GCS is claimed to transition to a passive role and monitor the mission as it unfolds. FlyMASTER was evaluated through software-in-the-loop (SITL) tests and an actual experiment involving a single drone. However, it does not provide control conditions and explicit drone synchronization.

It is important to note that the presented "Mission Syntax" is not necessarily the syntax used for an intermediate mission representation (if any), nor is it necessarily used for implementing any system components, such as the mission compiler. However, in EAMOS, we used the language Go for all of the aspects of mission description, mission intermediate form, implementation of the mission compiler, as well as for the implementation of one of the two main parts of mission execution, which is the Drone Execution Environment (the other part, the Mission Middle Layer is implemented in C++).

B. COMPARISON WITH EAMOS

Dragule et al. [40] extensively reviewed existing mission specification approaches, focusing on visual and end-user-oriented mission specification environments and the provided features of the individual approaches. They analyzed 30 approaches and identified 23 that have characteristics of mission specification environments with block-based languages. At the same time, 13 belong to purely text-based mission specification languages (such as domain-specific languages Aseba [41] and PROMISE [42]). They further identified C/C++, Python, JavaScript, and Basic as the most commonly used target languages to compile the mission specification languages.

Our analysis of the related work reveals that GUI-based approaches offer a better user experience than text-based approaches but tend to reduce drone missions to pure waypoint flights with little space for custom scenarios.

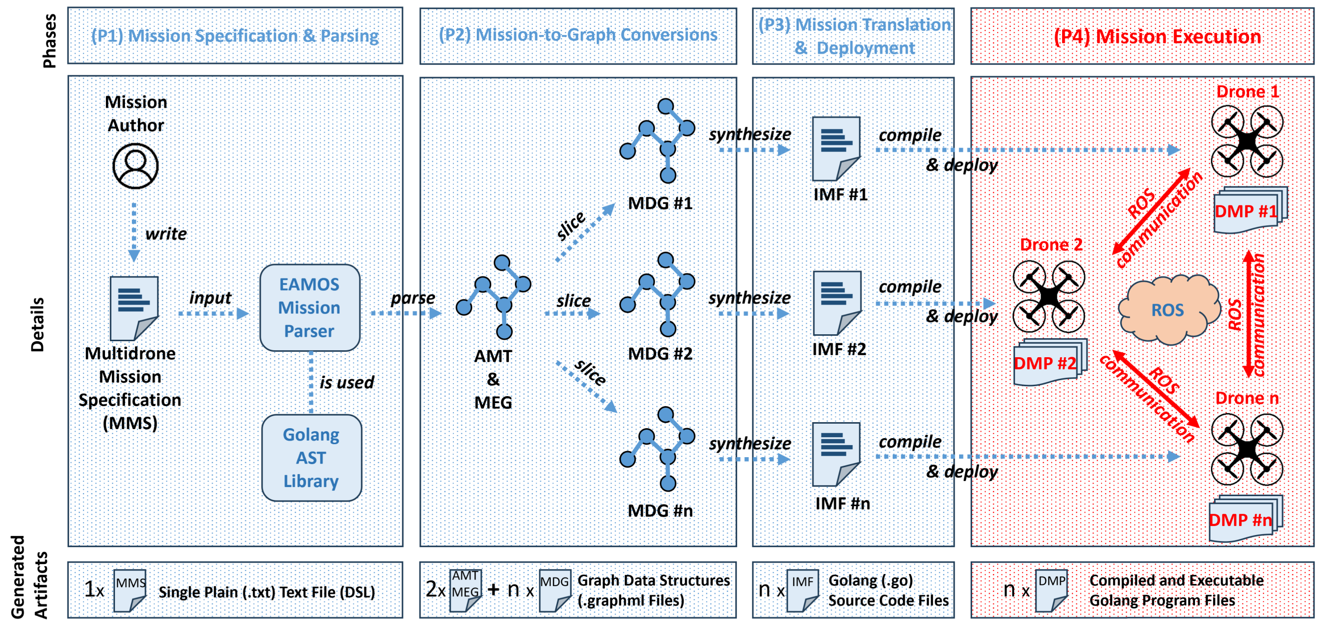

FIGURE 2. Overview of the applied methods in EAMOS. In Phase P1, the end user writes the multidrone mission specification (MMS) as Go-based text into a text file, which is then forwarded to the EAMOS Mission Parser. The parser uses functionalities from the Golang libraries to generate an abstract syntax tree. Phase P2 first generates one Abstract Mission Tree (AMT) and one Mission Execution Graph (MEG) (i.e., two .graphml files) from the overall multidrone mission. The MEG is then sliced into an individual Mission Dependency Graph (MDG) for each drone (i.e., multiple .graphml files), according to the drone's mutual dependencies that emerged from the multidrone mission. Phase P3 automatically synthesizes an Intermediate Mission File (IMF) (cf., Listing 4) from each MDG as plain Golang source code that is then compiled by the Golang compiler to executable Go programs and deployed as Drone Mission Programs (DMP) on the target drones. Phase P4 executes the drone-specific parts of the multidrone mission onboard the corresponding target drones, which use a ROS network for synchronization and data exchange.

On the other hand, many text-based approaches offer rather specialized capabilities, making them only applicable to a narrow range of use cases. While most approaches introduce their system architecture, the expandability to add or customize capabilities is neglected to a great extent. Allowing users to synchronize concurrent drone activities explicitly is a significant aspect of our work. While most approaches support multiple drones, most do not put the user in charge of synchronizing drone activities, or the respective tools perform the synchronization internally. A distributed execution architecture is another important aspect of our work that is not supported in most investigated approaches. Moreover, we did not find any indication that any approaches can handle simulated and real drones in the same multidrone mission (i.e., supporting hybrid, real, and virtual drone missions).

III. METHODOLOGY

The development and application of the EAMOS framework encompasses static and dynamic aspects. The applied methods can be grouped into the four successive phases **Mission Specification & Parsing (P1)**, **Mission-to-Graph Conversion (P2)**, **Mission Translation & Deployment (P3)** and **Mission Execution (P4)**. Each phase takes a set of files as input and produces a set of files as artifacts, which are handed over to the succeeding phase. Figure 2 illustrates the applied methods and the generated artifacts.

As described in Section VI-B and depicted in Figure 9, EAMOS uses the idea of the Communicating Sequential Processes (CSP) concurrency model [43], which essentially states that individual threads of execution synchronize and exchange data through inter-connecting channels. While Occam [44] is a famous but old example of a programming language using CSP for inter-process synchronization and communication, EAMOS uses the modern programming framework Go,[4] which builds upon the same CSP fundamentals. Go provides Goroutines, which are lightweight threads of execution managed by the Go runtime. Channels in Go have two terminals and are permanently attached to a variable on either side, which are typically hosted by different Goroutines. Channels are used to send data from one of their attached Goroutines to the other and block the receiving end if no data is sent through the channel. Thus, Go channels make maintaining concurrent implementations very simple and lightweight.

EAMOS' execution model is based on a distributed graph of drone actions that span over all involved drones, in which multiple actions execute simultaneously at different positions within the graph and automatically trigger other actions according to the multidrone mission. Since all drone actions are blocked threads that get unblocked by other actions during the mission, the underlying model is similar to classical Petri nets [45]. Classical Petri nets consist of

[4] https://go.dev/

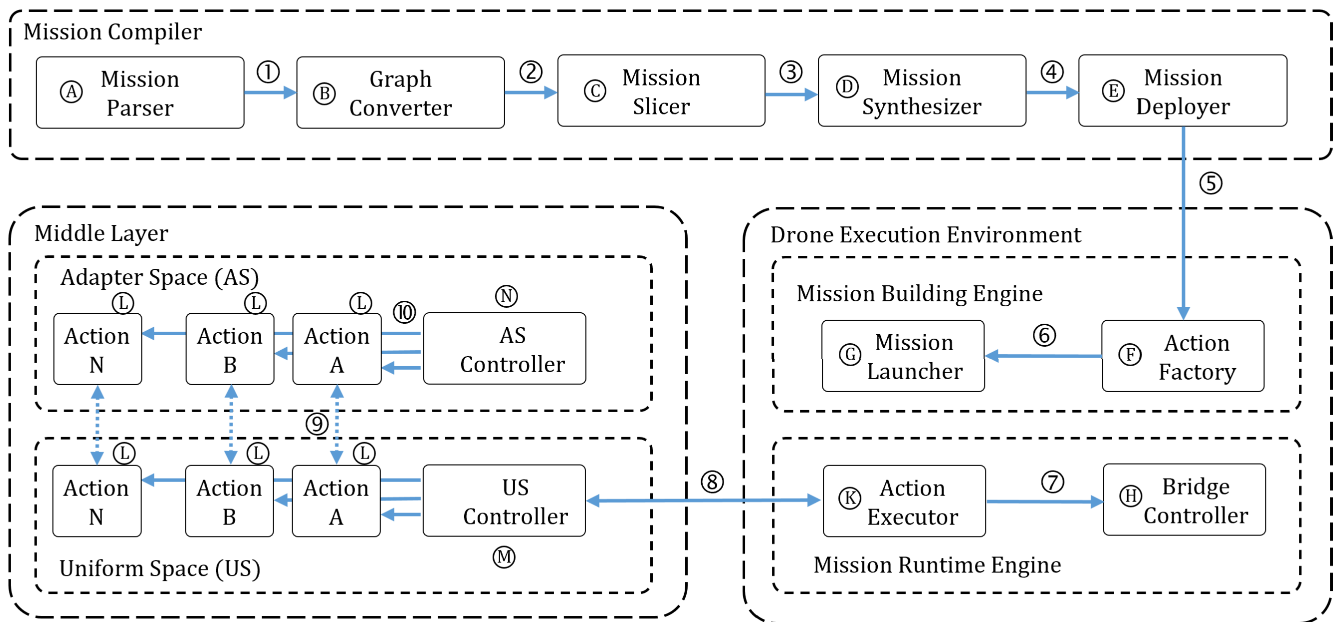

FIGURE 3. Overview of the components (black) of the EAMOS framework and their most important relations (blue). (A) parses multidrone missions from the user. (B) creates AMT and MEG. (C) creates MDGs. (D) creates IMF. (E) compiles and packs IMFs to deployment packages. (F) instantiates different objects that represent mission objects. (G) establishes all links among nodes and launches the initial mission node. (H) manages communication via the In- and Out-Bridge of a node to and from remote drones. (K) monitors preceding action nodes, performs the blocking, and triggers succeeding nodes. (M) provides the public API, instantiates, and controls US actions. (N) instantiates and controls AS actions. (L) implements US and AS actions as ROS action_lib server/client pairs. (1) Abstract syntax tree of the parsed mission forwarded to create graphs. (2) The MEG is forwarded to slicing (3). The MDGs for the drones are forwarded to synthesize intermediate programs. (4) The IMFs are forwarded for packaging. (5) Deployed drone programs are forwarded to create runtime objects. (6) Runtime objects are forwarded so that the mission can be launched. (7) Executor uses In- and Out-Bridges to interact with remote drones. (8) Executor uses Uniform Space to call actual actions. (9) US actions interact with their counterpart AS actions. (10) AS and US controller manage the lifecycle of their actions.

so-called transitions that move tokens through a network of so-called places. They represent frequently occurring patterns in the context of concurrent processes, such as simple sequences, resource (token-) conflicts, token multiplication (concurrency), process synchronization, or process (token-) merging. Many extensions to Petri nets were introduced, and the mathematical foundation is well established. We adopted colored Petri nets [46] to model action synchronization in EAMOS, but used a simpler notation where transitions and tokens are not modeled explicitly (cf., Section VI-B and Figures 9 and 11).

IV. FRAMEWORK OVERVIEW

EAMOS covers the complete processing stack of providing a mission specification environment, compiling specified missions into an intermediate representation, compiling and deploying these missions for specific target drone platforms, and executing compiled missions onboard drones. Figure 3 provides a high-level overview of the components of the EAMOS framework together with their most important relationships. While the Drone Execution Environment was implemented entirely using the language Go to take advantage of Go's concurrency model, we also used Go for implementing the Mission Compiler to make these components easily compatible and to simplify the processing of the mission specifications and their intermediate representations, which are also encoded in Go. On the other hand, we decided to implement the Middle Layer entirely in C++ because most ROS interfaces and tools use C++, which also significantly supports compatibility. While Listings 4 and 5 give an insight into the Action Factory component (cf., (F) in Figure 3), Listings 6, 7, 8, and 9 illustrate the key aspects related to mission execution of the Action Executor component (cf., (K) in Figure 3). Furthermore, Listings 2 and 3 give a glimpse of how drone actions manifest within the Drone Execution Environment and especially show the calling of actual drone actions from the public Middle Layer in Lines # 5 and 2 of these listings, respectively.

In general, any mission creation starts with writing a textual mission specification, which is inputted into the framework's Mission Compiler (MC). The MC converts the multidrone mission into an intermediate mission representation while also generating several graphs that reflect the structure and the execution flow of the mission. The MC is a stand-alone Go application that can be used on any machine, making it an off-board EAMOS component.

As the name suggests, the Drone Execution Environment (DEX) is in charge of executing an individual drone mission onboard a specific drone. The DEX is compiled and deployed with the drone mission within a single executable file that does not need installation or further configuration. Launching a drone's mission also launches its mission execution

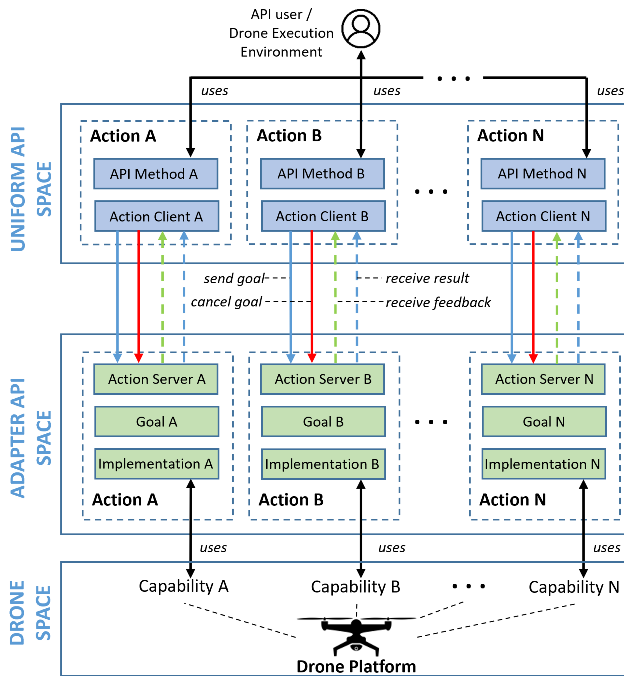

FIGURE 4. Illustration of EAMOS' Middle Layer showing the Uniform Space (US) and one Adapter Space (AS) for one drone platform. The US defines public API methods of drone actions that end users or the execution environment can use. Each uniform action has a counterpart adapter action. Both actions communicate through an action client-server pair provided by ROS' action_lib. Actions of the AS have a goal definition used by the action client of the action_lib communication to tell the server what it shall do. Action clients send or cancel a goal towards the server (blue and red solid arrows), while the server provides feedback for ongoing actions or results for finished actions (green and blue dashed arrows). AS actions further have a specific implementation that interacts with the drone and calls the actual capability needed for the US action called in the first place.

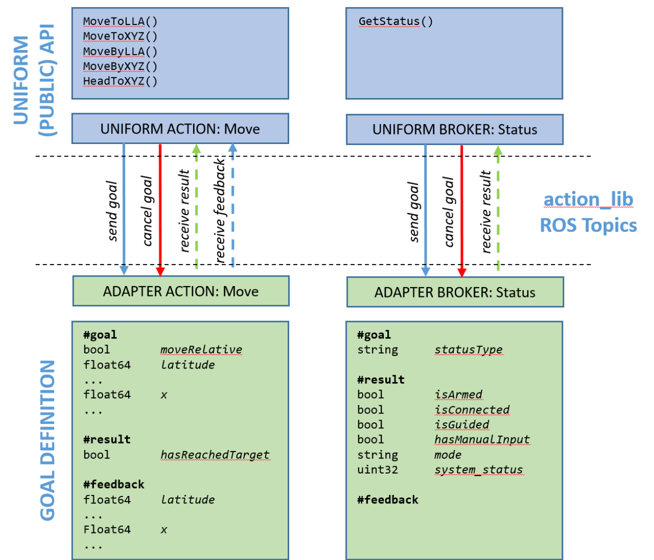

FIGURE 5. Illustration of the EAMOS action "Move" and the EAMOS Broker "Status" wrt. to Figure 4. The uniform Move action (left, blue box) defines several public API calls for moving drones (using either latitude, longitude, altitude, or local XYZ locations). The exemplified corresponding Move adapter action (left, green box) defines in its overall goal definition: goal–the format of the global LLA- and local XYZ-Location; result– a flag on whether the stated location was reached or not; feedback–the current location is continuously sent back until the target location was reached. Similarly, the uniform status action consists of a single public API call, which is expected to state which status to request. Depending on the kind of status, the Status adapter action returns several status values. This broker is a single-request-and-return action and thus provides no continuous feedback.

environment, which is embedded in the same binary file. Like the MC, the DEX is implemented entirely in Go, which supports compatibility among these two components.

EAMOS provides a middle layer tier (MID), which wraps the technical details of a drone's specific hardware interface into a uniform drone API, which is supposed to look and behave alike for every user within the framework. The MID contains a Uniform Space (US) that defines the generic uniform API and different adapters for different hardware platforms bound within an Adapter Space (AS). US and AS interact with each other by utilizing the action_lib library from the ROS ecosystem. This involves Action Servers and Action Clients that are attached to all individual drone capabilities and data providers. Using this design enables a loosely coupled mapping between uniform and specific API calls or data flows. The MID is entirely implemented in C++ to facilitate hardware compatibility.

Figure 4 illustrates the architecture of EAMOS' Middle Layer as follows. The Uniform Space (blue boxes) is primarily intended for drone action calls from within drone missions executed by the DEX. Due to its public API, it can also be used by any user (top users). It defines several public uniform actions that form EAMOS' public API and is thus common to all drones. Each uniform action provides a private action client, which interacts with its counterpart adapter action and its public action server. The Adapter Space (green boxes) on the drone side defines a publicly visible action server for each action and an individual ROS action_lib-Goal definition to specify the parameters for the action to execute. To trigger adapter actions, uniform actions define and send goals through their action client downwards (solid blue arrow). Goals can also be canceled while the adapter action is already working on it (solid red arrow). The ability to cancel already executing actions is vital for EAMOS' mission control structures (cf. Section VI-D).

The action server keeps returning a constant feedback (dashed green arrow) and a final result (dashed blue arrow) to the calling action client. The implementation of each adapter action interacts with the underlying drone hardware, whose set of individually provided and implemented capabilities form the Drone Space (bottom box). For simplicity, Figure 4 shows only the actions of an adapter for one drone platform. The EAMOS brokers work in a similar way to the actions.

Figure 5 shows how the public API and the public goal definition for an EAMOS action and a broker are defined by using the Move-Action and the Status-Broker. EAMOS' "Move" defines five public methods that can be used by any user (blue top box) as an example. Each call and its arguments

get converted into a goal defined by the corresponding adapter action. The arrows illustrate the communication (see Figure 4). The goal for "Move" either sets global (in GS84 format) or local locations (in NED format) to be reached, defines further that the constant feedback consists of the current location in both formats, and provides a success flag for reaching the goal. In contrast to triggering actions, a broker acquires data and provides it to its caller. EAMOS' "Status" broker has just a single uniform API method for getting the current status. Its adapter goal defines a set of information returned as a goal result. Since most broker executions are instant, no constant feedback is defined. ROS' action_lib library provides all communication between action clients and the action server. It manifests in the ROS environment as ROS topics, which could be used to interact with the action server.

Since the DEX uses API calls from the US and data is pushed back and forth between these two components during runtime, they must be compatible. To avoid introducing another interface or mapping component, EAMOS utilizes the SWIG[5] language interface framework to build a language bridge between DEX and US, letting the DEX directly call C++ methods from within Go-code and vice versa without any cross-compilation.

V. MISSION COMPILER

EAMOS' Mission Compiler is responsible for reading a single textual multidrone mission specification and converting it into a form the framework onboard drone platforms can execute. Since this conversion is done prior to executing a mission, mission compilation is considered to belong to the framework's off-board processing. The overall purpose of the compilation is to extract all components from the multidrone mission together with their intra- and inter-drone dependencies (also referred to as internal and external dependencies) and to only deploy them to drones to which the dependencies are relevant.

Mission compilation undergoes the consecutive stages of parsing a multidrone mission specification (Section V-A), constructing the Abstract Mission Tree and the Mission Execution Graph, which both reflect the overall multidrone mission (Section V-B), extracting drone-relevant components and dependencies for every drone in the form of separate Mission Dependency Graphs (Section V-C), creating high-level human-readable code files that represent individual drone missions (Section V-D) and deploying these drone missions as executable programs to their dedicated drones (Section V-E).

A. MISSION PARSING

To support simplicity and readability, multidrone mission specifications conform to standard Go syntax. This also enables the mission author to utilize inbuilt Go language features for parsing Go source code files and constructing and working with abstract syntax trees (AST). Mission Listing 1 shows an example of an EAMOS multidrone mission specification for two drones performing several parallel and sequential actions, such as turning on and performing a preflight check. These mission files are parsed using inbuilt libraries of the Go framework to create an AST, which represents the hierarchy of the mission in terms of its nested Go function calls and that further consists of all the mission's Go statements.

[5]https://www.swig.org/

```
1  package DemoMission1 //declare mission by name.
2
3  var Drone1 DroneType1 //declare Drone1 as Type1.
4  var Drone2 DroneType2 //declare Drone2 as Type2.
5  var Drone3 DroneType3 //declare Drone3 as Type3.
6
7  //is triggered automatically.
8  func INIT() {
9    SEQ_Start()}
10
11 //launch children sequentially.
12 func SEQ_Start() {
13     PAR_TurnOn() //first this.
14     PAR_PreFlightChecks() //then this.
15     SEQ_ActivateCamera()} //then this.
16
17 //launch children in parallel.
18 func PAR_TurnOn() {
19     Drone1.TurnOn() //launch with other two.
20     Drone2.TurnOn() //launch with other two.
21     Drone3.TurnOn() //launch with other two.}
22
23 //launch children in parallel.
24 func PAR_PreFlightChecks() {
25     SEQ_PreFlight1() //first this.
26     SEQ_PreFlight2()} //then this.
27
28 //launch children sequentially.
29 func SEQ_PreFlight1() {
30     Drone1.PreFlightCheck() //first this.
31     Drone1.Calibrate() //then this.}
32
33 //launch children sequentially.
34 func SEQ_PreFlight2() {
35     Drone2.PreFlightCheck() //first this.
36     Drone2.Calibrate() //then this.}
37
38 //launch children sequentially.
39 func SEQ_ActivateCamera() {
40     Drone2.TurnOnCamera() //first this.
41     Drone2.AdjustCamera() //then this.}
```

LISTING 1. Illustration of a simple three-drone mission specification. "SEQ"-prefixes indicate sequential execution flows, and "PAR"-prefixes indicate parallel execution flows. In this example, all three drones are turned on in parallel. Then, Drone1 and Drone2 perform a pre-flight check followed by a calibration in parallel. Finally, Drone2 first turns on its camera and then adjusts it. light-blue–particular drones to call actions on; dark-blue–drone types; light-orange–drone actions; green–comments; dark-orange–Go keywords and syntax.

B. MISSION GRAPH PROCESSING

An integral part of the mission compilation is constructing the internal mission representation, which describes drone actions, control structures, execution branches, and the dependencies among drone actions in the form of directed graphs. The first graph produced during mission compilation is the Abstract Mission Tree (AMT), which is then converted to the Mission Execution Graph (MEG).

1) ABSTRACT MISSION TREE

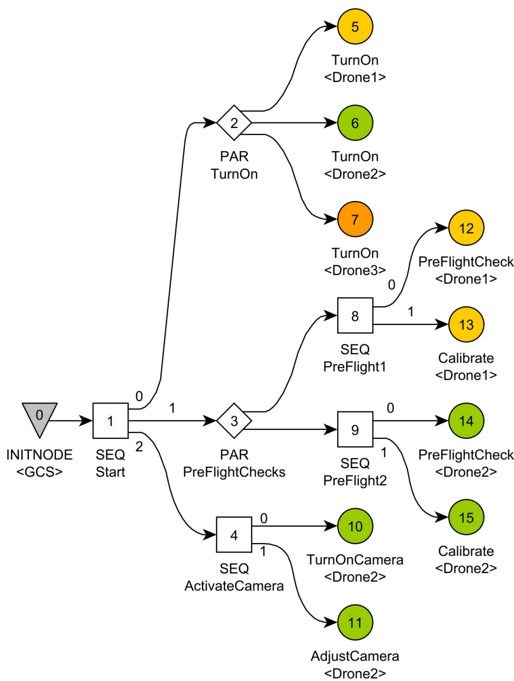

FIGURE 6. Abstract Mission Tree of the multidrone mission in Listing 1. squares–sequential executions; diamonds–parallel executions (i.e., forks); triangle–initial mission node; circles–drone actions. Node indices are shown within nodes. Children of sequential nodes are numbered to reflect their sequential execution order. grey–initial node; white–not yet assigned to a drone; yellow–Drone1; green-Drone2; red–Drone3.

The Abstract Mission Tree (AMT) reflects the structure of the multidrone mission as a tree whose leaves are actions, whose non-leafs are either sequential branches, parallel branches, or conditions, and whose root is the initial mission node. Parallel branching nodes result from parallel blocks within the multidrone mission and are named forks. The initial node is a unique and required node artificially added to the mission representation as a predecessor of the first mission statement of the multidrone mission.

Figure 6 depicts the AMT of our simple three-drone mission. Labels for nodes provide information about the name and the type (condition-type or action-type). Labels at sequential branches indicate the order of the subsequent nodes. Since no order is implied for parallel branches, labeling is omitted for these branches. Action and condition nodes are already assigned to drones at this stage while branching nodes are assigned to nodes during the slicing phase of the mission compilation (cf. Section V-C).

2) MISSION EXECUTION GRAPH

The Mission Execution Graph (MEG) reflects the dynamic execution flow of the multidrone mission as a directed graph, whose syntax is the same as for an AMT. The key difference is that sequential executions of actions are now depicted as chains of actions reflecting the true sequence of actions. This makes dedicated sequential branching nodes (depicted as squares in the AMT) obsolete. The MEG results from what we refer to as "closing" the sequential nodes of the AMT or "closing" the AMT, respectively. The term "closing" refers to the process of connecting leave nodes of the AMT according to the imposed execution order so that the resulting directed graph has no "open" leaves anymore, except those that terminate the overall mission. Figure 7 shows the MEG of Listing 1.

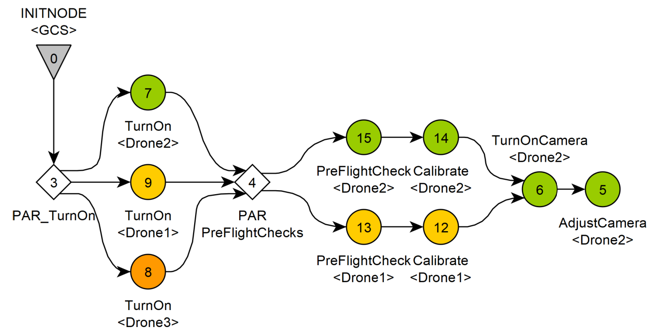

FIGURE 7. Mission Execution Graph that corresponds to the multidrone mission in Listing 1. diamonds–parallel executions (i.e.; forks); triangle–initial mission node; circles–drone actions. Node indices are shown within nodes. In contrast to the AMT, the MEG has no sequential nodes anymore (cf. squares in the AMT in Figure 6). The only dangling node in the MEG is node 5 ("AdjustCamera", <Drone2>), which terminates the multidrone mission. grey–initial node; white–not yet assigned to a drone; yellow–Drone1; green-Drone2; red–Drone3.

C. MISSION SLICING

Mission Slicing marks the stage where individual mission graphs for every drone are generated from the overall multidrone mission. These drone mission graphs, referred to as Mission Dependency Graphs (MDG), consist of all action nodes of a particular drone and all dependencies of these nodes to and from other internal and external nodes. Since all drone operations defined by nodes within an MDG are eventually to be physically executed onboard a particular drone, all of their nodes are required to get the appropriate associated drone. This is a non-trivial procedure because some structural and conditional nodes, such as forks, need to be replicated for different drones, thus becoming so-called node proxies.

A node proxy is any node that is a replicated clone of another node that physically belongs to and exists onboard another drone. An example is node 1 ("n1", <Drone2>) in Figure 8 (c), which has the two node proxies node 2 ("n2 Fork", <Drone1>) and node 2 ("n2 Fork", <Drone3>), since they are adjacent but on remote drones. Here, Drone2 has two replicated copies of these node proxies in its MDG. Similarly, node 0 ("n0", <GCS>) has node 1 ("n1", <Drone2>) as a node proxy. This relationship is also represented by external arcs and indicated by dashed lines. Here, the GCS has a replicated copy of its node proxy in its MDG.

Forks are more challenging to replicate than non-forks for two reasons: First, forks might have multiple children that can be assigned to different drones, which means that

Algorithm 1 EAMOS' Slicing Algorithm. For Each Drone of the Multidrone MEG, a Separate MDG Is Created and Each MEG Node That Corresponds to That Drone Is Processed by Three Consecutive Steps

Input: MEG m
Output: MDG_{Drone_1}, MDG_{Drone_2}, $\ldots$
1: **for all** distinct drones d occurring in MEG **do**
2: create empty MDG_{Drone_d} m
3: **for all** nodes n in MEG **do**
4: **if** n is assigned to drone d **then**
5: **for all** successors s of n **do**
6: ForwardForkProcessing(n, m)
7: **end for**
8: CurrentNodeProcessing(n, m)
9: **for all** predecessors p of n **do**
10: BackwardForkProcessing(p, m)
11: **end for**
12: **end if**
13: **end for**
14: **end for**

Algorithm 2 Forward Fork Processing Algorithm: Probe All Initially Reachable Non-Forks From a Node n and Create for All of Their Distinct Drones a New Fork-Proxy for the New MDG That Is Connected From the Current Node n. Link(p, n) Links Node p to Node n With an Internal or External Arc Depending on Whether They Have the Same Drone Assigned

Input: current node n, MDG g for drone d
1: **for all** succeeding forks s of n **do**
2: starting at s as root and considering non-forks as leafs: apply DFS and store all encountered leafs in L
3: **for all** distinct drones d' that are assigned to leafs in $L \setminus d$ **do**
4: create new fork f' and assign it drone d'
5: Link(n, f')
6: **end for**
7: **end for**

all of them need to exist as native nodes and as proxy nodes. Second, forks can be nested within each other, forming multi-level hierarchies of forks resulting from nested parallel function calls in the multidrone mission specification. EAMOS processes forks through forward and backward algorithms to tackle these challenges, yielding an overall three-step procedure for slicing a MEG into its corresponding MDG.

The idea of the procedure (see Algorithm 1) is to iterate over each node of the MEG wrt. the drones involved and to apply the three steps described below depending on whether the current node's predecessors and successors are either forks or not. Figure 8 shows an example MEG and its resulting MDG for Drone1, Drone2 and Drone3.

1) STEP 1: FORWARD FORK PROCESSING

This step aims to determine what we call fork-proxies for action nodes. Fork-proxies are any nodes that are no forks by themselves but are immediate successors of forks, which follow the respective action node. Identifying fork successors for action nodes is trivial if an action node is succeeded by not more than a single fork in a row. However, it becomes more challenging when considering hierarchies of forks consisting of Fork 2, Fork 3, and Fork 4 in Figure 8 (a). Forward fork processing is illustrated in Algorithm 2.

The MDG of Drone2, for example, must provide the information which successor node needs to be triggered once node 1 (''n1'', <Drone2>) finished executing. Now, when looking at Figure 8 (a), we see that node 1 (which belongs to Drone2) is supposed to trigger all six leaf nodes of the MEG in parallel, whereas some of them are internal to Drone2 and some are external to it. Thus, the MDG of Drone2 needs to link its native node 1 to internal and external successors for every external drone affected by node 1 (the so-called node proxies). If node proxies are forks rather than action nodes, we need to introduce a separate node proxy for every external node the fork is pointing to. This is where fork proxies come into play. From all the fork proxies of a particular fork, we consider one representative for each different drone and attach it as a node proxy to the preceding node of the fork considered in the first place.

Figure 8 (c) shows the MDG of Drone2, and here, node 1 points to a node-proxy representing all fork-proxies of Drone1 (the orange Fork 2) as well as to one node-proxy representing all fork-proxies of Drone3 (the red Fork 2). The green successor Fork 2 is no proxy, but a local successor that results from processing step 3 described further below.

2) STEP 2: REGULAR NODE PROCESSING

This step handles a current node's neighboring non-fork nodes (i.e., all predecessors and successors). It links them together using internal or external arcs depending on whether the current node and its particular neighbor are on different drones. Regular node processing is illustrated in Algorithm 3.

The MEG in Figure 8 (a) only has one pair of neighboring non-fork nodes, which are node 0 (''n0'', <GCS>) and node 1 (''n1'', <Drone2>). Thus, the MDG for Drone2 needs to consider node 0 as external and node 1 as internal. This can be seen in Figure 8 (c) by the dashed arc from node 0 to node 1 (external arc) and the solid arc from node 1 to fork 2 (internal arc).

The MDG for the GCS (not shown here) only consists of the initial node 0 and node 1, which are connected by an external arc from node 0 to node 1 and by considering node 1 as the external node.

3) STEP 3: BACKWARD FORK PROCESSING

This step aims at assigning the appropriate drone to the path of all preceding forks of a particular current node, which goes

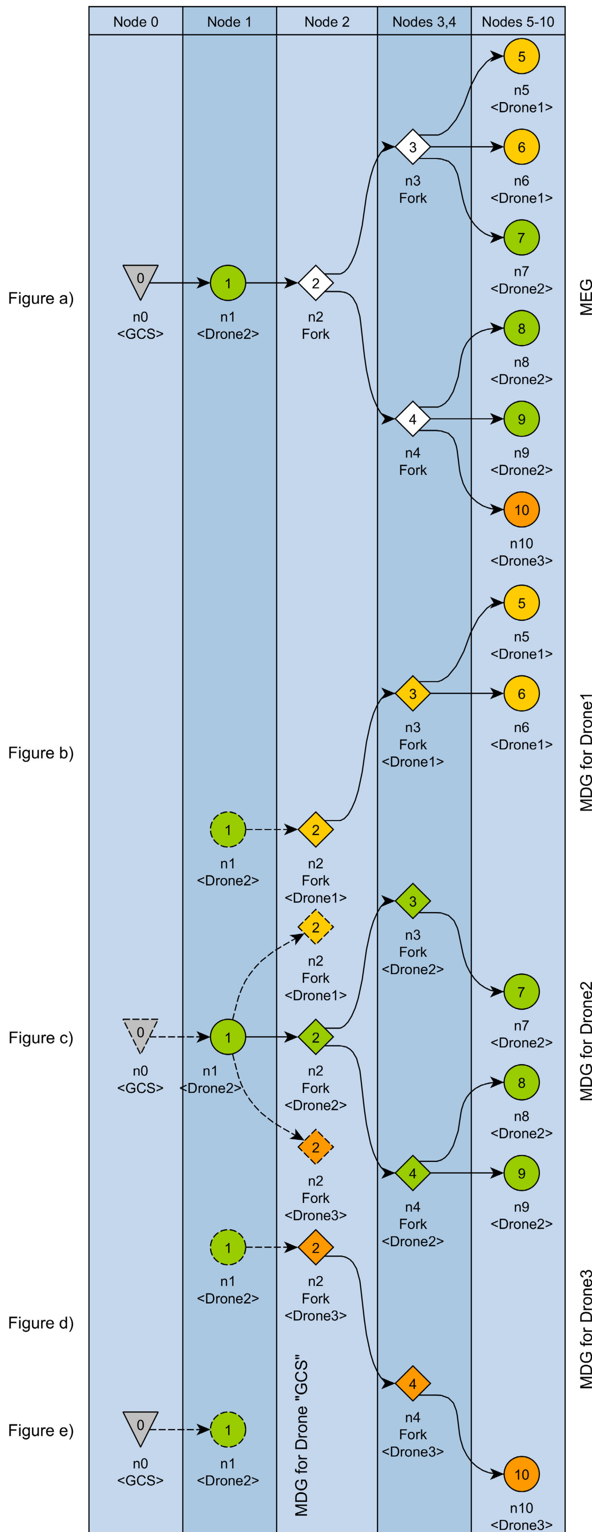

FIGURE 8. Example of another mission illustrated by its Mission Execution Graph (a) involving three drones and their corresponding Mission Dependency Graphs: (b) MDG Drone1; (c) MDG Drone2; (d) MDG Drone3; (e) MDG Drone "GCS"; grey–initial node; white–not assigned to a drone yet; yellow–Drone1 (b); green–Drone2 (c); red–Drone3 (d). External links (and nodes) are drawn with dashed lines, and internal ones have solid lines.

Algorithm 3 Current Node Processing Algorithm: Link All Non-Fork Predecessors and All Non-Fork Successors of a Non-Fork n Together by External or Internal Arcs

Input: current node n, MDG g for drone d
1: replicate n and add it to g
2: **for all** succeeding non-forks s of n **do**
3: Link(n, s)
4: **end for**
5: **for all** preceding non-forks p of n **do**
6: Link(p, n)
7: **end for**

all the way backward to the first non-fork node. Backward fork processing is illustrated in Algorithm 4.

Considering, for example, node 5 or node 6 from Figure 8 (a) as current nodes (while constructing the MDG for Drone1), we assign preceding Fork 3 and Fork 2 to Drone1, which is taken from node 5 (see Figure 8 (b)). If this is done for node 5, node 6 already has properly assigned preceding forks, and no new backward processing is performed. The same backward processing assigns Fork 3 and Fork 2 to Drone2 because of node 7, Fork 4 to Drone2 because of either node 8 or node 9 (see Figure 8 (c)), and Fork 4 and Fork 2 to Drone3 because of node 10 (see Figure 8 (d)).

Algorithm 4 Backward Fork Processing Algorithm: If a Non-Fork n Has Fork Parents, Iterate From Parent to Parent Until a Fork Has No More Fork Parents. Every Fork of This Chain Is Replicated for the New MDG and the Parents of the Last Fork Are Linked With It

Input: current node n, MDG g for drone d
1: **while** parent p of n is of type fork **do**
2: replicate p and add to g
3: link p to n (int. arc)
4: $n = p$
5: **end while**
6: **for all** non-fork predecessors p' of n **do**
7: Link(p', n)
8: **end for**

D. MISSION SYNTHESIS

After generating Mission Dependency Graphs, the next step is to convert each MDG into a form that can be executed onboard the target drone. For this purpose, EAMOS utilizes the programming language Go and its execution environment for several reasons.

First, EAMOS' Mission Execution Environment is entirely implemented in Go, which provides a high degree of compatibility between the execution environment and the drone missions to be executed. Second, Go is a modern, syntactically lean, feature-rich programming language, making it very suitable as an intermediate representation for

EAMOS' drone missions. Third, Go's runtime environment is lightweight and well-supported on different embedded computer platforms, which makes it ideal for our use onboard the companion computers of drones.

The core of this compilation stage is synthesizing a Go source code file for each MDG, referred to as Intermediate Mission File (IMF). This file represents the drone missions as a human-readable and easy-to-understand text file, which is eventually compiled by the Go compiler to an executable Go binary that executes the drone mission.

The structure of the IMF is separated into a declaration and an execution part. The declaration part implements two significant aspects. First, each node of the MDG is instantiated as a mission object that provides individual properties and capabilities (see Listings 2 and 3). Second, all mission objects are linked according to their connections within the MDG. Internal links connect two local mission objects, while external links connect remote mission objects wrt. the drone they are assigned to. The Mission Execution Environment implements links using Go-channels, which belong to the conceptual core of EAMOS' execution model.

Listing 4 shows a snippet of the IMF for Drone2 that corresponds to the MDG in Figure 8 (c). Lines #1-#11 instantiate mission objects that correspond to the graph nodes of the MDG. The arguments of the (orange) factory methods, such as CreateAction0, are as follows: argument 1 is the drone this object originally belonged to; argument 2 defines the textual identifier; argument 3 specifies the actual drone action that gets executed when this mission object is triggered; and argument 4 specifies the predecessor-mission object. While the signatures of factory methods differ wrt. the types of mission objects, the drone that hosts the drone action built by the corresponding factory method and the predecessor argument are part of the declaration of every mission object.

Once all mission objects are instantiated, line #15 triggers a linking algorithm that uses the predecessor of every mission object to link all internal mission objects to a consistent graph structure. External links from a remote origin to a mission node are implemented by attaching the object to the In-Bridge (#line 18), and external links from a mission object to a remote target are handled by attaching the object to the Out-Bridge (#line 19). Finally, all mission objects get activated, enabling them to execute automatically when all their predecessors have finished executing (line #23).

E. MISSION SETUP

The last step of the mission compilation is the mission setup, which prepares all file assets and arranges all required files in the necessary directory environment for later mission rollout. The most important file assets are:

1) **Drone Execution Environment**: This bundle of Go source code files comprises all logic needed to execute individual drone mission files onboard drones. The files are arranged together to be compiled later.

```
1 func (drone *Drone) FlyByXYZ(x string, y string, z
     string) {
2   x_f64, _ := strconv.ParseFloat(x, 64)
3   y_f64, _ := strconv.ParseFloat(y, 64)
4   z_f64, _ := strconv.ParseFloat(z, 64)
5   DroneInterface.FlyByLocalDistance(drone.Name,
      x_f64, y_f64, z_f64)}
```

LISTING 2. Example definition of an IMF drone capability using a Middle Layer drone interface capability. Colors indicate: light-orange–name of the capability that is attached to the particular drone (cf. drone actions in Listing 1); light blue–the drone object for which the capability is defined; dark blue–drone type; green–the Go-interface object that was generated by SWIG and which provides access to the Uniform Space API of EAMOS' Middle Layer; red–method from the Uniform Space API of the Middle Layer; purple–input parameters for this capability; dark-orange–Go keywords and syntax.

```
1 func (drone *Drone) LLAPosition() (float64,
      float64, float64) {
2   longitude, latitude, altitude := DroneInterface.
      LLA(drone.Name)
3   return longitude, latitude, altitude
4 }
```

LISTING 3. Example definition of an IMF drone property using a Middle Layer drone interface property. Colors indicate: light-orange–name of the property that is attached to the particular drone; remaining colors as in Listing 2.

2) **SWIG Interface Environment**: This file bundle specifies how the drone missions interface with the underlying EAMOS Uniform Space. The bundle consists of the SWIG interface file (defining some type conversions between EAMOS mission and EAMOS Uniform Space), the EAMOS Uniform API header file (the file that defines the important EAMOS Uniform API), and two building scripts that support the building during the later mission rollout.

The result of this stage is the EAMOS Mission Setup Space, which is ready to be copied to the external mission compiler and compiled to eventually produce executable drone mission files (cf. Section VI-E).

VI. DRONE EXECUTION ENVIRONMENT

EAMOS' Drone Execution Environment (DEX) is in charge of instantiating and managing mission nodes, linking them within drones and among drones, and executing them through managing their inter-dependencies. These tasks are encoded to be done by the IMF as exemplified in Listing 4, which corresponds to the MDG of Drone2 in Figure 8 (c). Considering that the variable "Drone" in Listing 4 represents the Drone Execution Engine for a particular drone, the three tasks above are described as follows.

Create methods such as CreateAction0 or CreateFork instantiate new mission nodes. The last argument of these Create-Methods simply states all predecessors of the corresponding node. After calling these methods for each node, the whole structure of the MDG can be resembled. The methods AttachRcvActionExecution and AttachSndActionExecution attach remote nodes to In- and Out-ROS bridges, respectively, which realizes the linking of mission nodes across remote

```
// mission object instatiations.
INITNODE_GCS = *(Drone.CreateInit(GCSName,"INITNODE"))
n1_Calibrate = *(Drone.CreateAction0(D2Name,"Calibrate",D2.DoCalibrate,&INITNODE_GCS.Base))
n2_FORK_Drone3 = *(Drone.CreateFork(D3Name,"FORK_Drone3",&n1_Calibrate.Base))
n2_FORK_Drone1 = *(Drone.CreateFork(D1Name,"FORK_Drone1",&n1_Calibrate.Base))
n2_FORK = *(Drone.CreateFork(D2Name,"Fork",&n1_Calibrate.Base))
n3_FORK = *(Drone.CreateFork(D2Name,"Fork",&n2_FORK.Base))
n4_FORK = *(Drone.CreateFork(D2Name,"Fork",&n2_FORK.Base))
n7_PreFlightCheck = *(Drone.CreateAction0(D2Name,"PreFlightCheck",D2.DoPreFlightCheck,&n3_FORK.Base))
n8_ActivateSensor1 = *(Drone.CreateAction0(D2Name,"ActivateSensor1",D2.DoActivateSensor1,&n4_FORK.Base))
n9_ActivateSensor2 = *(Drone.CreateAction0(D2Name,"ActivateSensor2",D2.DoActivateSensor2, &n4_FORK.Base))

// internal linking of all native mission objects.
for _, a := range allActions {
  a.Link()}

// external linking of remote mission objects.
D2Node.AttachRcvActionExecution(&n1_Calibrate.Base)
D2Node.AttachSndActionExecution(&n1_Calibrate.Base)

// setting every native mission object into a ready-to-be-executed state.
for _, a := range droneActions {
  go a.Activate()}
```

LISTING 4. **Intermediate Mission File snippet for Drone2. The IMF encodes a particular MDG and is automatically synthesized by EAMOS. Colors indicate: blue–mission objects instantiated through the IMF; red–utility objects that provide mission object instantiating and that are provided by the IMF environment; light-orange–factory methods to first instantiate different mission objects and to then link and activate them; purple–drone action as provided by EAMOS' Middle Layer; dark-green–mission object identifiers; light-green–code comments; dark-orange–Go keywords and syntax.**

drones. Once the Activate method activates mission nodes, they are ready to launch when their prerequisites are satisfied automatically.

A. GO-CHANNELS

Since EAMOS' execution model relies on Go channels, the following briefly describes their basic application. Go channels can either be bidirectional (as declared by `channel chan string`)[6] or unidirectional (as declared by `channel chan<- string` or `channel <-chan string`) and either read and send data from a variable (as in `channel <- x`)[7] or assign a received value to a variable (as in `x <-channel`). Channels transfer data from one end to the other (i.e., from one thread to another) and block the receiving thread until data is received, which brings the sending thread in charge of blocking and releasing the receiving side. The throughput of channels can be limited (i.e., buffered channels), and the `select` clause allows to listen to multiple incoming channels simultaneously. These and many more lightweight features of Go's programming model allow us to realize the interactions among drones through an intuitive and appropriate computational model. EAMOS even extends the concept since it uses channels to effectively interconnect physical machines by attaching Go channels to ROS topics (cf., Section VI-C) rather than just local threads.

B. EXECUTION MODEL

EAMOS' execution model follows the simple concept that any mission node automatically starts executing its embedded drone action when all its preceding mission nodes have finished executing. Once a mission node finishes executing, it triggers all its successors, ultimately performing a controlled chain reaction of executions among all activated nodes. The core idea to realize this scheduling of executions is using uni-directional Go-channels to either block or trigger the execution of mission nodes. By writing a multidrone mission, any mission node's preceding and succeeding nodes are implicitly defined by specifying their order and embedding them in sequential or parallel execution blocks. Preceding nodes are referred to as node preconditions, and succeeding nodes are referred to as node consequences. Preconditions are connected to their target node through in-going Go-channels, and any node connects to its node consequences through outgoing channels.

1) LINKING

Each mission node in EAMOS monitors all in-going Go-channels and keeps track of how many have already signaled the execution of their preceding mission node on the other side of the channel. Once all in-going preconditions of a mission node are satisfied (i.e., all preceding mission nodes have finished executing), the node executes its mission action. Then, it triggers all of its consequences (i.e., signaling all successive mission nodes). Thus, two mission nodes are interconnected by a Go-channel if the execution of one node depends on the execution of the other node. This ultimately builds an execution network in which mission nodes can have any number of preconditions and consequences and which are always in a defined state, either describing untriggered, partially triggered, or triggered nodes (cf. Figure 9).

The linking procedure performed at drone mission startup (cf. Listing 4 line #15) establishes the execution

[6] `channel` is the name of a channel, `chan` is the Go channel keyword
[7] `channel` is a channel name and `x` is any variable.

```
func (base *Base) linkByPredecessors() {

  for _, pre := range base.predecessors {
    if pre.DroneName == base.DroneName {
      c := make(chan Package)
      base.ingoing = append(base.ingoing, &c)
      pre.outgoing = append(pre.outgoing, &c)
    }
    pre.successors = append(pre.successors, base)
  }
}
```

LISTING 5. Linking method that establishes execution interconnections between a node and its predecessors. Colors indicate: light blue–the node that is currently linked; dark blue–the base struct for that node; purple–the set of in-going and out-going channels of the current node; green–the set of predecessors and successors of the current node; orange–Go keywords and utilities.

interconnections of a mission node by creating a new Go-channel for each of its predecessors and installs these channels between a new out-going-port of each predecessor and a new in-going-port of its own (cf. example the interconnection C6 between N0 and N3 in Figure 9, which adds N0 as precondition to N3 and N3 as a postcondition to N0). Listing 5 shows this simplistic linking algorithm.

2) EXECUTION

Mission node execution involves the three simple steps of (1) waiting until all preconditions are satisfied (cf. Listing 7), (2) executing the actual drone action, and (3) triggering all preconditions (cf. Listing 8). The first step is started by initially activating the mission nodes (cf. Listing 9). Once a node has been activated, an infinite loop takes care that it repeats, starting its execution whenever all preconditions are satisfied. The ability of a node to repeat its executions is essential for supporting mission control structures such as RepeatWhile or DoUntil.

EAMOS realizes the idling or blocking of a mission node until its preconditions are satisfied through Go-channels, which simply block or unblock the corresponding thread of execution. Although Go supports simultaneous listening to multiple channels and automatic handling of incoming data, the number of channels needs to be known at compile-time. The number of preconditions of a mission node (and thus Go-channels) not only varies from node to node but is also unknown for the execution engine without a particular drone mission to compile (i.e., during compile-time of the execution engine). Thus, a unique channel-selection mechanism that enables the automatic execution of a variable number of channel-select branches is necessary. The corresponding algorithm extends Go's standard channel-selection mechanism by using Go's reflection utilities and is shown in Listing 6. In this listing, the node execution states mentioned before are reflected by the number of channels within the cases slices in line #3, which contains all precondition channels of a node (node state: untriggered), containing any number of channels between all and one (node state: partially triggered), and containing none channels (node state: triggered).

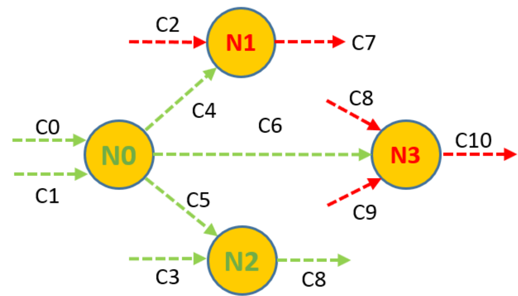

FIGURE 9. Example of four mission nodes, which are either fully or partially triggered. For example, for the fully triggered node N0, all in-going channels C0 and C1 (green in-going arcs) signaled that their attached preceding node finished executing, making N0 unblocking (green labeled node), triggering its successors N1, N2, and N3 by out-going channels C4, C5, and C6 (green out-going arcs). Similarly, node N3 blocks (red labeled node) because its in-going channels C8 and C9 did not yet signal the execution of their attached nodes (red in-going arcs) while having its in-going channel C6 (in-going green arc) being signaled.

C. GO-ROS BRIDGES

While pure Go-channels can only be used to realize the execution-blocking and triggering of mission nodes within a single drone, EAMOS extends the scope of Go-channels by connecting them with ROS topics to go beyond the edge of drone platforms. For this purpose, each drone execution environment provides an In-Bridge and an Out-Bridge, a Drone Service Provider, and a Drone Service Client. The latter's responsibility is to either subscribe to ROS services and connect their service providers with in-going Go-channels or to publish the payload forwarded by outgoing Go-channels into ROS service clients. Figure 10 illustrates the interplay of a drone's Out-Bridge and its Service Client to forward the triggering of mission object executions from one drone to another using the embedding ROS network.

This approach differentiates between mission nodes with dependencies to local neighbors onboard the same drone and those with dependencies to mission nodes on other drones. Both nodes use Go-channels to handle pre- and postconditions, but the latter group is attached to either the drone's In- or Out-Bridge. An outgoing channel receives the name of its mission node as a payload. It is attached to the local drone's Out-Bridge, which provides ROS service clients that forward the data signal to the subscribed ROS topic by the target drone. Since all drones are in the same physical ROS network, data and signals are sent back and forth among drones following the same execution model as if they were onboard a single drone. Figure 11 illustrates how mission nodes of one drone are attached to their In- and Out-Bridges to enable them to receive execution triggers from remote drones and send triggers to remote drones.

D. MISSION CONDITIONS

To realize a conditional execution flow through missions, EAMOS provides Mission Control Structures, which are syntactic constructs that provide conditions, loops, and

```
1  func (base *Base) SelectChannels(channels []*chan Package, action func(/* ... */), params *Base) {
2
3    cases := make([]reflect.SelectCase, len(channels)) // prepare a slice for all select cases needed.
4
5    for i, channel := range channels { // initially, iterate over all channels.
6      cases[i] = reflect.SelectCase{ // generate dynamic select cases for each channel.
7        Direction:  reflect.SelectRecv, // declare, that we want to read from the channels.
8        Channel: reflect.ValueOf(*channel), // assign the actual channel to the current select-case.
9        }
10     }
11
12   for len(cases) > 0 { // now, start iterating over all dynamically created select-cases.
13     chosenChan, rcvdValue, _ := reflect.Select(cases)  // this is, where we actually block; Here, Go's
       reflect utility monitors all channels for incoming data and activates the corresponding select-case
       if data arrives; That's the same mechanism as hard-coding Select-clauses in code for a constant
       number of cases, just for a variable number of cases.
14
15     // we only reach here, if a select case was chosen to execute because data arrived in its
       corresponding channel.
16
17     lock.Lock() // we protect case-executions if multiple channels get data simultaneously to avoid race
       conditions.
18
19     package := extractPackage(&rcvdValue) // extract data from the channel.
20     action(len(channels), package.OriginAction, params) // perform an action that is dedicated just to
       satisfying this particular drone node precondition.
21     cases = RemoveSliceElement(cases, chosenChan) // remove the used select-case.
22
23     lock.Unlock() // leave the critical region.
24     continue // loop back to the blocking state where we wait for any other channel to receive data.
25   }
26     // we reach here, when all channels received data and thus all preconditions were satisfied; This
       mission node is now free to trigger its postconditions next.
27 }
```

LISTING 6. Code snippet that implements monitoring a drone node's in-going Go-channels and blocking the node if not all preconditions are satisfied. This code extends Go's standard Select-clause functionality to receive from various channels simultaneously. Colors indicate: green–the passed set of channels and the currently iterated channel, respectively; blue–created dynamic select-cases; purple–utilities of Go's reflect-library; light-orange–channel and data once it sends data; dark-orange–Go keywords and syntax.

```
1 func (base *Base) pre(ingoing []*chan Package) {
2   base.SelectChannels(ingoing, action, base)
3 }
```

LISTING 7. Method for entering the automatic channel-reading of all in-going channels of the mission node. Colors indicate: blue–the mission node; green–set of in-going channels coming from predecessors; light-orange–the automatic and dynamic channel selection procedure (cf. listing 6); purple–type of data that is passed through the channels; dark-orange–Go keywords and utilities.

```
1 func (base *Base) post(outgoing []*chan Package) {
2
3   for i := range outgoing {
4       *outgoing[i]
5         <- Package{
6           Origin: base,
7           Destination: &Base{Name: ""}}
8       }}
```

LISTING 8. The method that triggers the postconditions of a mission node by sending data into the node's out-going channels. Colors indicate: green–a set of out-going channels into which data is sent; blue–the mission node; purple–type of data sent into the channels; orange–Go keywords and utilities.

waiting mechanisms in various forms. Table 3 provides an overview of currently implemented control structures. Though being integrated into EAMOS' set of features, the ability of the Do-control structure to interrupt executing

```
1 func (action *Action2) Activate() {
2
3   for {
4     action.pre(action.ingoing)
5     action.execute(action.param0(),action.param1())
6     go action.post(action.outgoing)
7   }
8 }
```

LISTING 9. The Method that initially activates each mission node. Once activated, execution runs into an infinite loop of waiting until all preconditions are satisfied, executing the actual drone action, and triggering the postconditions as a separate thread of execution. Colors indicate: light-blue–action that is activated; dark-blue–action type (declaring a drone action with two parameters); light-orange–methods that wait until all preconditions are satisfied and that trigger all postconditions; green–set of in-going and out-going Go-channels of the action; red–method that encapsulates the actual drone action using the parameters of the action; dark-orange–Go keywords and syntax.

actions is not yet implemented and will be added at a later development stage.

E. MISSION ROLL-OUT

Mission roll-out refers to all steps involving physical file transfers among computers (copy- and deploy steps), the compilation of the multidrone mission (compilation step), and the building of binaries (build step). Since the drones we used for our experiments (cf. Section VII-D) have a

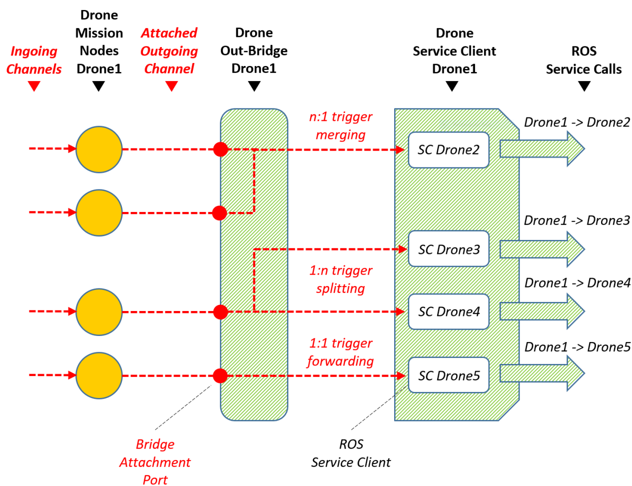

FIGURE 10. **Illustration of how Go-channels of a drone's (Drone1) local MDG are connected to ROS-topics to enable communication with remote drones (e.g., for triggering remote mission nodes). Mission nodes that trigger remote mission nodes (yellow nodes) are attached to the drone's Out-Bridge (red dots), which forwards the messages to be sent to the drone's service client, where they get fed into ROS-topics that are sent to the remote ends of Drone2, Drone3, Drone4, and Drone5 by triggering ROS-service calls. The Out-bridge is also in charge of correctly distributing the node's out-going channels to appropriate individual service clients by merging, splitting, or forwarding messages.**

Raspberry Pi companion computer, compilations currently target the ARM64-architecture. We use a small external Raspberry Pi as an external compilation unit to supersede the need to cross-compile from an x64 architecture (since mission editing is most conveniently done on a desktop or notebook computer). Mission roll-out is triggered by the user on demand whenever the mission changes.

1) **Copy Mission**: This step transfers the multridrone mission source file to the external compilation unit by a simple file transfer (via WiFi) and initially sets up the remote directory environment needed for the next roll-out steps.
2) **Compile Mission**: On the compiling unit, the multidrone mission is then compiled by the EAMOS Compiler following all steps described in Section V.
3) **Build Mission**: On the compiling unit, once the synthesized individual drone mission source code files from the previous compilation stage are available, the files are compiled by the Go-Compiler to produce executable Go binary files. Moreover, the SWIG-Compiler is used to join the drone mission files with the EAMOS Uniform Space files during mission compilation to produce a compound executable drone mission binary for each target drone of the multidrone mission (SWIG Compilation only needs the Uniform Space C++ header files, because for usability reasons the Uniform Space is assumed to be already compiled and available as C++ binaries).
4) **Deploy Mission**: The final step is the physical file transfer of the compiled individual drone mission files (just a single file per drone) to the target drones (via WiFi).

F. MISSION LAUNCH

Since EAMOS is embedded in a ROS environment, launching missions onboard drones requires some subsystems to be up and running. Thus, the mission launch involves the following steps and systems to be launched onboard each drone. All following steps are automatically launched with a single executable batch file for ease of use.

1) **Sourcing ROS Environment**: Initially, all involved ROS packages need to be sourced, and several environment variables must be set for the mission to run.
2) **Launch MAVProxy**: We use the tool MAVProxy[8] that receives MAVLink[9] packages from the Pixhawk flight controlling unit (FCU) and forwards it to whoever needs them. We forward them to the MAVROS program and the monitoring tool QGroundControl.[10]
3) **Launch MAVROS**: We use MAVROS[11] to wrap MAVLink messages into ROS topics, which makes the FCU (its sensor data, its telemetry, and its capabilities) available within our ROS environment and thus for our ROS-based EAMOS Adapter Space.
4) **Launch EAMOS Adapter Space**: Each drone's individual adapter must run onboard the drone. The adapter receives mission action requests from the EAMOS Uniform Space, which is embedded in the drone's mission file and processes the requests toward the drone's hardware.
5) **Launch Mission**: The actual drone mission incorporates EAMOS's Uniform Space and runs together once launched.
6) **VRPN Client (optional)**: Since we conducted indoor flights, a tool was necessary to receive position data from the motion capturing system and wrap it into ROS topics that MAVROS can read. This tool is optional if conducting outdoor flights since the localization is provided through satellite data.

G. MISSION TRIGGER

An EAMOS multidrone mission is launched by starting all individual drone missions. This brings them into an initial ready state, where each drone waits for its first action trigger according to the multidrone mission. EAMOS introduces a "Ground Control Station" (GCS) as a virtual drone and adds an initial mission node as the first mission statement to each mission. The multidrone mission is launched by sending triggers from the GCS drone to all initial mission nodes.

[8]https://firmware.ardupilot.org/Tools/MAVProxy/
[9]https://mavlink.io/en/
[10]http://qgroundcontrol.com
[11]http://wiki.ros.org/mavros

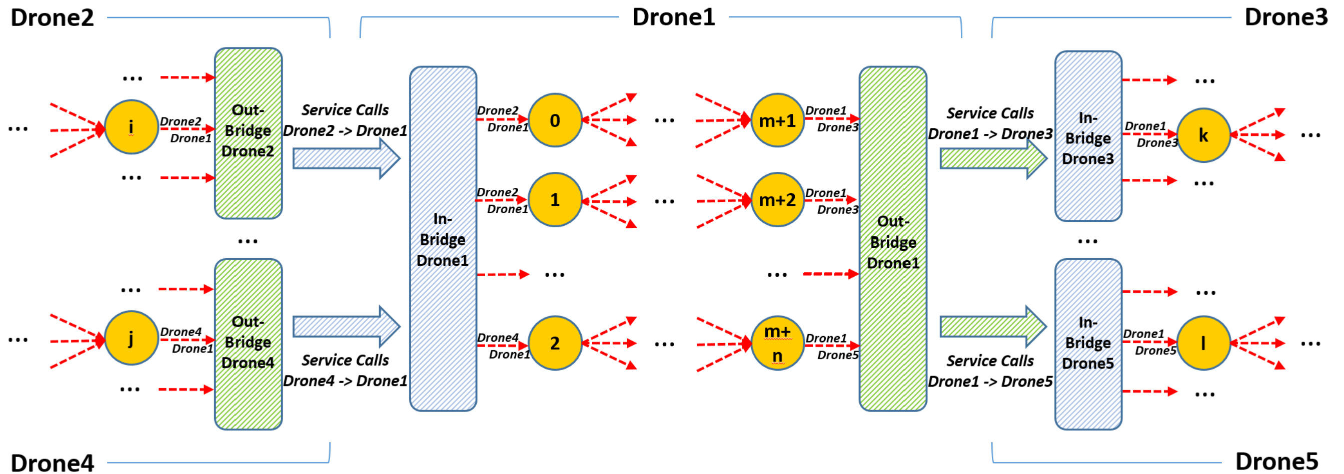

FIGURE 11. Illustration of the communication (e.g., triggering of mission nodes) among four remote drones, Drone2, Drone3, Drone4, and Drone5, with a local drone, Drone1 using In- and Out-Bridges. Red arcs are Go-channels, blue arcs are received service calls, and green arcs are sent service calls. While Out-Bridges simply pack the triggering information into service calls and send it to the corresponding remote end, In-Bridges receive remote service calls, read the included node triggering information and forward it to the corresponding local node.

TABLE 3. Overview of Mission Control Structures in EAMOS.

Name	Form	Description
If		Simple If-Condition in the If-Then-Else form.
Wait	While	Wait for execution of one/many action(-s) **while** a certain condition is true.
	Until	Wait for execution of one/many actions(-s) **until** a certain condition becomes true.
Repeat	While	Repeat execution of one/many action(-s) **while** a certain condition is true.
	Until	Repeat execution of one/many action(-s) **until** a certain condition becomes true.
	For	Repeat execution of one/many actions(-s) **for** a constant number of repetitions.
Pause	While	Pause mission execution **while** a certain condition is true.
	Until	Pause mission execution **until** a certain condition becomes true.
	For	Pause mission execution **for** a constant duration.
Do	While	Execute one/many action(-s) **while** a certain condition is true (and interrupt if it becomes false).
	Until	Execute one/many actions(-s) **until** a certain a condition becomes true (and interrupt if it becomes true)

VII. EXPERIMENTAL SETUP

A. EXPERIMENTS

To demonstrate the applicability and correct functionality of our EAMOS framework, we conducted four experiments[12] (experiments A, B, C, D) that highlight different aspects of the framework. The experimental scenarios range from simple to complex and target both single drone (experiment A) and multidrone missions (experiments B, C, D). To emphasize the abstraction and generalization capabilities of EAMOS' Middle Layer to support heterogeneous drone platforms, we set up two hybrid experiments (experiments C, D) that involve two real drones and one simulated drone in the same multidrone mission. Furthermore, experiment D involves the most complex scenario, through which we demonstrate EAMOS' mission control structures by setting up an interactive (as well as hybrid) multidrone mission, which enables the user to influence mission execution manually.

We also tested EAMOS by executing missions using Microsoft's Airsim[2] in its 2023 version simulation environment, which provided us with a highly accurate and appropriate drone vehicle and environment simulation. Our ROS-based and platform-agnostic architecture allowed us to easily use the Software-in-the-loop approach, where we ran the same flight controller firmware on our simulated drones as on the real quadcopters. Moreover, the EAMOS drone mission programs could be deployed and executed in the simulation similarly to the real drones.

Table 4 summarizes the experiments, and Table 7 provides an overview of the results of all experiments. Section VIII presents details of the four experiments, explaining the purpose, the mission, and the results.

B. EVALUATION

We examined the following mission aspects to analyze the experiments and evaluate the performance.

1) **Mission Compilation Duration**: Mission compilation is part of mission roll-out and comprises the actual mission compilation and its stages of parsing, graph processing, slicing, synthesizing, and setup (cf. Section V).
2) **Mission Deployment Duration**: Mission deployment refers to the physical file transfers of the compiled

[12]A video of the experiments is available at https://youtu.be/JfVlbXMdh90.

drone deployment packages from the external mission compiler (the external Raspberry) to the target drones.

3) **Total Roll-Out Duration**: In addition to compilation and deployment duration, the complete mission roll-out comprises the file transfers to the external mission compiler, the setup of remote directories, and the removal of potentially existing files.
4) **Assessment of Mission Execution**: To assess whether the drones performed exactly as specified by the corresponding mission, the movements of the drones during the experiments were tracked and recorded by a motion tracking system, mirrored into several detailed log files, and observed by multiple human testers.
5) **Number of MDG Nodes**: Each node of a drone's Mission Dependency Graph is synthesized to an executable mission statement in the Go drone mission file by the EAMOS' Mission Synthesizer (cf. lines 2-11 in Listing 4).
6) **Number of internal MDG Arcs**: Internal arcs of a drone's MDG represent local actions onboard the same drone that are linked together by Go-channels (cf. Listing 5 and Figure 9).
7) **Number of external MDG Arcs**: Action triggers from remote drones are represented by in-going external arcs and are dispatched through EAMOS' In-Bridge. In contrast, triggers to remote drones are represented by outgoing external arcs and are dispatched through EAMOS' Out-Bridge (cf. Figure 11). Both bridges use ROS service calls and EAMOS' Service Clients and Service Providers for the underlying communication (cf. Figure 10).
8) **Mission File Size**: The file size of the single binary mission file, which is the main deliverable of the mission roll-out for a particular drone, is executed to start the drone's mission onboard that drone.
9) **ROS Service Calls**: As mentioned before, action triggers from a remote drone or to a remote drone are realized by ROS service calls, where the number of received service calls of a drone needs to match the number of in-going external arcs of its MDG, and the number of sent service calls needs to match the number of outgoing external arcs of its MDG.

Table 6 provides an overview of the evaluated aspects and from where their values are obtained.

C. DATA ACQUISITION

To consistently track all activities during mission execution, EAMOS maintains three kinds of detailed log files that make the internal processes traceable and documentable. In contrast, every drone involved in the mission maintains its own log files to have a clear separation of concerns regarding drone activities:

1) **Mission Log File**: Logs high-level mission activities, such as internal action nodes triggering other nodes, external node triggers passing through the In- and Out-bridges by receiving and sending service calls, and monitoring the state of pre- and post-conditions of action nodes.
2) **Middle Layer Uniform Log File**: Logs which actions were triggered by a particular drone's mission file and how they were passed further to the Adapter Space of the drone.
3) **Middle Layer Adapter Log File**: Logs all activities of EAMOS' Adapter Space onboard a particular drone, which makes the actual execution of mission actions visible and includes interesting sensor information such as drone locations.

Furthermore, the experiments were video recorded to document the exact development of each experiment. Since experiments B and C involve a simulated drone, the screen that showed the virtual drone in its simulated environment (as well as the real drones in the background) was filmed simultaneously to the camera that primarily filmed the real drones. Finally, the "motion takes" from Optitrack's[13] Motive[14] were also recorded and precisely showed the movements of the real drones in the drone hall. Table 5 provides an overview of EAMOS' data acquisition during the experiments.

D. ENVIRONMENT

We conducted our experiments at dronehub Klagenfurt[15] which provides a large drone hall with a flight volume of more than 1300 m^3. Drone motions can be precisely captured with a 360 Hz Optitrack[13] motion tracking system composed of 37 "Prime17W" infrared-detecting cameras (cf. Figure 12).

EAMOS was evaluated indoors because the experiments were more manageable and safer to carry out and observe. Since the drones were secured by thin ropes in the drone hall, all waypoints of the missions were within a few meters of the starting positions. EAMOS can be applied for outdoor missions without any modifications; only drone positioning must be changed from motion capturing to GPS positioning, which is the standard position method for most drones.

To set up and perform our four experiments consisting of single drone, multidrone, and hybrid scenarios, we used a setup essentially consisting of the following components (cf. Figure 13):

1) two Pixhawk4[16]/PX4-equipped quadcopters with an on-board Raspberry-Pi v4 companion computer
2) a mission notebook for setting and monitoring missions remotely onboard individual drones
3) a desktop computer for executing the SITL drone simulation with its 3D-realistic Airsim[2] simulation (for experiments C and D)
4) a Raspberry-Pi v4 computer in an external cooling case, used for ad-hoc compiling and deploying missions

[13]https://optitrack.com/
[14]https://optitrack.com/software/motive/
[15]https://uav.aau.at
[16]https://docs.px4.io/v1.12/en/flight_controller/pixhawk4.html

TABLE 4. Overview of the experiments conducted to demonstrate the applicability of the EAMOS framework. # Actions: total number of actions called from EAMOS' Uniform API (number of Pause-Actions used for evaluation stated in parentheses). # Par- or Seq-Blocks: total number of sequential and parallel execution blocks in the mission. Conditional: states whether the mission involves mission control structures. Hybrid: states whether simulated drones are involved in the mission.

Experiment	Mission Type	# Drones	# Actions	# Par/Seq Blocks	Conditional	Hybrid
A	Single-Drone	1	16 (8)	0 / 1		
B	Multidrone	2	36 (11)	1 / 7		
C	Multidrone	3	36 (4)	1 / 8		X
D	Multidrone	3	14 (0)	11 / 2	X	X

TABLE 5. Overview of the data acquisition during the conducted experiments to document the results.

Data Source	Maintainer	Scope
Video Footage	Video Cameras	Cam 1: Real Drones
		Cam 2: Virtual Drone
Mission Log File	Drone	Action Triggering
		State of Pre-/Post-Conditions
		In-/Out-Bridge-Invocations
Uniform Log File	Drone	Uniform Action Invocations
Adapter Log File	Drone	Adapter Action Invocations
		Drone Locations
		ROS Topic Subscriptions
		ROS Action Goal states
		Processing of Adapter Actions
Motive Motion Takes	Optitrack Motive	Movements of real drones

TABLE 6. Overview of the aspects that were evaluated wrt. each conducted experiment.

Aspect	Obtained via	Unit
Mission Compilation Duration	Log Files	min:sec
Mission Deployment Duration	Log Files	min:sec
Total Roll-Out Duration	Log Files	min:sec
Mission executed Correctly	Observation & Data	Yes / No
# Nodes of MDG	MDG	integer
# int. Arcs of MDG	MDG	integer
# ext. Arcs of MDG (in / out)	MDG	integer
Mission File Size	Examining File	kB
Mission Execution Duration	Video Footage	min:sec
# ROS Service Calls (rcv / snd)	Mission Log File	integer

5) two remote controls for the drones to be able to take over manual control for safety reasons
6) a Microsoft X-Box-controller to manually control the simulated drone (for experiment D)
7) the Optitrack[13] desktop computer running Optitrack's Motive[14] GUI for motion capturing
8) the WIFI-infrastructure of the drone hall

E. DRONES

For our experiments, we used two identical quadcopters, "twinFOLD SCIENCE v2" customized by "twins GmbH" (see Figure 14). The length of one rotor arm is 414 mm and the overall beam has a length of 520 mm. Its maximum take-off-and landing weight is 1700 g while using a 4-cell battery with 14.8 V and 5500 mAh. The used propellers had three blades with a 153.07 mm diameter, a 4.7 in incline, and a weight of 6.46 g. The propellers were actuated by four brushless "T-MOTOR F40 PRO V FPV" motors with 1950 KV.

Each drone was equipped with a Holybro Pixhawk 4[16] flight controlling unit (FCU) and an electronic speed controller (ESC) of type "FlyColor X-Cross BL-32 / 35A". We used the most recent PX4[17] firmware (v1.13.3) for the Pixhawk boards. Additionally, each drone was equipped with

FIGURE 12. The drone hall of the dronehub Klagenfurt has 150 m^2 ground area, 10 m height, and the 1300 m^3 capturing volume. A protective cage in the background protects the operators while conducting drone flights [Daniel Waschnig (AAU)].

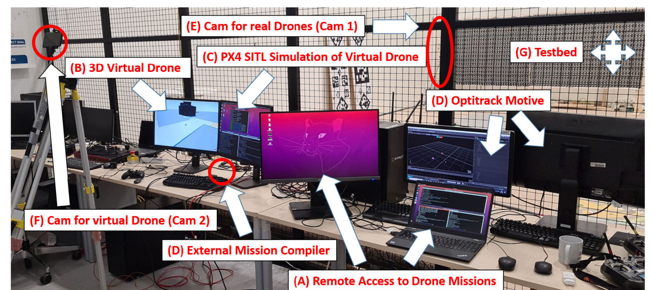

FIGURE 13. Overview of the computer systems we used in our experimental setup. One notebook with two screens (A) was used to remotely access the Raspberry Pi computers onboard the drones to monitor diverse drone states and to launch the missions. A desktop computer showed the 3D real-time position data of the drones by Optitrack's Motive on two screens (D). Another desktop computer ran Airsim and the 3D virtual drone simulation on one screen (B) and the attached PX4 software-in-the-loop flight controller simulation on another screen (C). An external Raspberry Pi was used for mission compilation (D). One camera (F) filmed the virtual drone in the foreground and the real drones in the background. Another camera (E) was placed next to the real drones (G).

[17]https://px4.io/

FIGURE 14. twinFOLD SCIENCE quadcopter equipped with a mounted battery (center, red), GPS antenna (with blue LED light), antenna for telemetry (left, small ant.), WiFi antenna (left, long ant.), and one IR-reflective marker for the mocap-system (right, small gray bulb).

a Raspberry Pi 4 B onboard computer that was linked to the FCU by a USB-to-serial TTL adapter, whose source was the TELEM2 port of the Pixhawk FCU with a baud rate of 1000000 b/s. The onboard computer had WiFi connectivity and ran Ubuntu 20.4.05 LTS (focal fossa). To increase WiFi connectivity for the flights, each drone was equipped with an external WiFi antenna.

VIII. EXPERIMENTS

A. EXPERIMENT: SINGLE DRONE MISSION

1) PURPOSE

Experiment A aims to test and evaluate the roll-out and execution of a single drone mission with one real drone. The compilation was performed on our external EAMOS mission-compilation unit (i.e., the external Raspberry Pi) and took approximately 2-3 minutes for the overall roll-out (including file transfers over WiFi).

The mission is a simple fly-to scenario in which the drone takes off and flies to four positions that describe a rectangular flight path before it lands at its starting position. Without loss of generalization, any mission actions can be used to demonstrate EAMOS' capability to correctly execute both single- and multidrone missions by synchronizing action executions wrt. sequential or parallel action arrangements within a mission. Hence, the scenarios in the experiments described here use simple fly-to actions in their missions.

2) MISSION DESCRIPTION

In the following, Listing 10 shows the EAMOS mission specification for experiment A, and Figure 15 shows its Mission Execution Graph. To clearly interpret the drone's movements for human observers, we let the mission pause for five seconds between each mission action using the EAMOS Pause-action (i.e., PauseFor(5)). We removed these Pause-actions from the mission specification and the mission graphs shown in this section to not degrade readability.

3) RESULTS

The results of experiment A are summarized in Table 7 in column "A". The presented values include 8 Pause-actions of the mission executed in experiment A, which were not included in the listing, and the figures of this section to favor readability. Since this is a single drone mission, the only external in-going arc and received service call is the initial trigger from the ground control station to begin the mission.

B. EXPERIMENT: TWO-DRONES MISSION

1) PURPOSE

The purpose of experiment B is first to demonstrate full mission roll-out (i.e., file transfers, compilation, setup, deployment) and, in particular, the execution of an EAMOS multidrone mission involving two real drones. This scenario is more complex than experiment A because it involves sequential and parallel executions. As stated before, the kind of the actual actions is not essential for demonstrating correct mission execution, but the synchronization of the drones among themselves wrt. the sequential and parallel action executions as specified in the multidrone mission is of paramount importance for demonstrating EAMOS' multidrone capabilities.

2) MISSION DESCRIPTION

After both drones have armed themselves one after another, they take off in parallel to a height of 1.5 m. Once both drones reach their take-off height, both keep performing simultaneous flight maneuvers by starting with a simultaneous flight for 1 m into positive X-direction. Once both have reached their target points, both fly simultaneously 1 m into positive Y-direction. Once they have reached their target points, both fly 1m into negative X-direction. Finally, when both have reached their target points, both fly 1 m in negative Y-direction. After reaching their target points, they hover above their respective starting points. As a final sequential maneuver, one drone ascents by 0.5 m, and when this position has been reached, the other drone performs the same ascent. When both drones hover at a height of 2 m, both start their landing descent simultaneously and disarm themselves after landing simultaneously. Listing 11 shows the multidrone mission specification, Figure 16 shows the Mission Execution Graph for the multidrone mission, and Figures 17 and 18 show the Mission Dependency Graphs for both drones.

3) RESULTS

The results of experiment B are summarized in column B in Table 7. The presented values include 11 Pause-actions of the mission executed in experiment B, which were not included in the listing, and the figures of this section to favor readability. Note that the number of external in-going

arcs of each drone's MDG matches the number of received service calls of each drone and that the same holds for outgoing external MDG arcs and sent service calls per drone.

C. EXPERIMENT: THREE-DRONES HYBRID MISSION

1) PURPOSE

The purpose of experiment C is to demonstrate the capability of EAMOS and its Middle Layer, respectively, to incorporate different drone platforms within the same multidrone mission. To emphasize EAMOS' possibilities, we chose a hybrid experiment involving a virtual drone (Drone3) and two real ones (Drone1 and Drone2).

In this experiment, the PX4 flight controller of Drone3 is fully simulated in a Software-in-the-Loop setup, while Microsoft's Airsim simulator 3D realistically simulates the drone itself. Both simulation environments and the deployed EAMOS drone mission for Drone3 run on a separate desktop computer, which is connected to the same WIFI-network as the other real drones and thus acts as if it would be another physical drone.

2) MISSION DESCRIPTION

In this scenario, Drone1 and Drone2 fly along the same rectangular flight path as in experiment B, but their movements depend on the movements of the virtual Drone3. Drone3 takes off and flies a $5\,\text{m} \times 5\,\text{m}$ rectangular path. While arming and take-off for all drones happen both in parallel and sequentially, respectively, at mission start, the real drones' individual movements only start when Drone3 finishes its appropriate movement. This is achieved by simply positioning the separate movements of Drone3 before the parallel movements of the real drones within a sequential block in the mission. In the end, all drones land and disarm in parallel. Listing 12 shows the mission specification for the scenario, Figure 19 shows the Mission Execution Graph, and Figures 20, 21, and 22 show the Mission Dependency Graphs for both drones.

3) RESULTS

The results of experiment C are summarized in column C in Table 7. As expected, it was irrelevant for the EAMOS framework that one drone was virtual and two were real. The multidrone mission[18] and their sliced individual drone missions controlled the corresponding drones they ran onto, independent of whether it was a real or simulated drone. As in the previous experiments, the number of received and sent service calls matches the number of in-going and outgoing MDG arcs of each of the three drones.

[18] Due to the Pause-actions inserted into the mission, the virtual drone was too long in the offboard mode and performed a short landing-and-launch maneuver between mission actions. However, the overall mission execution was not affected by this unspecified behavior.

D. EXPERIMENT: THREE-DRONES INTERACTIVE HYBRID MISSION

1) PURPOSE

This most complex scenario aims to demonstrate the two EAMOS mission control structures, Repeat-While and Wait-Until, and how they can provide a simple conditional execution flow. To make the scenario even more expressive, we added the virtual Drone3 to the mission and connected it with an additional remote control, which a person operated during the experiment. The idea of the mission is now that the location of the manually controlled Drone3 affects the movements of the two real drones. This added interactivity to the mission because the real drones were directly affected by the virtual drone, which was directly affected by the human operator, who indirectly affected the real drones.

2) MISSION DESCRIPTION

The mission involves the real drones Drone1 and Drone2 and the simulated Drone3. While the real drones fly in our drone hall, the virtual drone flies in a 3D-simulated environment, providing four square-shaped areas on the ground that are clearly separated by a couple of meters. Each of these squares now affects the real drones' movements differently if the simulated drone hovers over them. The squares are clearly marked and labeled as "Drone1 X+", "Drone1 X−", "Drone2 X+" and "Drone2 X−". If Drone3 hovers over square "Drone1 X+", the real Drone1 flies 20 cm into positive X-direction, and if Drone3 hovers over square "Drone1 X−", Drone1 flies 20 cm into negative X-direction, which results in a controlled movement of Drone1 of flying back and forth in the drone hall. The same holds for the other two squares wrt. the real Drone2.

Listing 13 shows the mission specification for the scenario, and Figure 23 shows the Mission Execution Graph.

3) RESULTS

As in experiment C, the ensemble of virtual and real drones within the same multidrone mission worked faultlessly. In particular, the interplay of Drone3 with the real drones through the mission control structures resulted in the expected movements of Drone1 and Drone2.

Table 7 in column D summarizes the results of experiment D. This time, the number of in-going and outgoing external arcs of the MDG of the drones does not match the number of received and sent service calls because due to the interactive character of the mission, the dynamic runtime of the mission deviated from its static specification.

During mission execution, the manually remote-controlled Drone3 flew over square "Drone1 X+" and stayed there so long that Drone1 flew three times in positive X-direction (3 service calls from Drone3 received by Drone1), followed by flying to the square "Drone1 X−", where it stayed so long, that Drone1 flew two times back in negative X-direction (2 service calls from Drone3 received by Drone1). Next, Drone3 flew to square "Drone2 X+" to let Drone2 fly two

TABLE 7. Results of the experiments A-D, presented per drone and showing all relevant evaluation criteria as described in Table 6. Yellow, green, and orange represent the drone-related results for Drone1, Drone2, and Drone3.

		Experiments			
Aspect	Drone	A	B	C	D
Scope		Simple Single-Drone Mission	Simple Multidrone Mission	Hybrid Multidrone Mission	Interactive Hybrid Multidrone Mission
Mission Compilation Duration		00:10 s	00:16 s	00:33 s	00:59 s
Mission Deployment Duration		00:11 s	00:18 s	00:19 s	01:24 s
Total Roll-Out Duration		00:36 s	00:37 s	01:28 s	03:12 s
Mission Executed correctly		Yes	Yes	Yes	Yes
# Nodes of MDG	1	18	29	26	20
	2		39	26	28
	3			45	34
# int. Arcs of MDG	1	17	11	9	6
	2		25	9	5
	3			15	19
#ext. Arcs of MDG (in / out)	1	1 / 0	10 / 7	8 / 8	6 / 4
	2		7 / 9	8 / 8	5 / 5
	3			15 / 12	6 / 7
# ROS Service Calls (rcv / snd)	1	1 / 0	10 / 7	8 / 8	5 / 9
	2		7 / 9	8 / 8	4 / 9
	3			15 / 12	9 / 11
Mission File Size	1	9 242 kB	9 252 kB	8 720 kB	8 716 kB
	2		9 248 kB	8 720 kB	8 711 kB
	3			8 764 kB	8 752 kB
Mission Execution Duration		01:21 s	01:20 s	02:12 s	04:02 s

times in positive X-direction, followed by flying to the square "Drone2 X−" to let Drone2 fly back two times in negative X-direction (both squares resulted in 2 service calls each from Drone3 received by Drone2).

E. STRESS-TEST

Moreover, we investigated the scalability of the EAMOS mission compiler wrt. the size of the mission specification. While experiments A–D missions were compiled and executed on embedded Raspberry Pi platforms, we conducted a separate stress test on a standard desktop computer. We used the Go profiler pprof[19] to asses the runtime and memory requirement, firstly, of the EAMOS mission compiler during compilation, and secondly, of the runtime environment during the start of a multidrone mission.

We assessed two missions: a small single-drone mission with a 34-node MEG and a huge 20-drone mission with more than 5 000 nodes. Compilation of the first mission required 4 368.78 kB of memory and took 50 ms. Launching this mission for Drone1 (with 22 nodes) required 3 749.50 kB of memory and took 916.63 ms. Compiling the second mission required 23 979 kB of memory and took 2.34 s. Launching the mission on a randomly selected drone (with an MDG with 634 nodes) out of the 20 drones required 4 257.45 kB and took 733.64 ms. Since each drone, once it received its deployed drone mission program, executes independently in an action-by-action manner on its own, we conclude from this stress test that the EAMOS space and runtime requirements scale well and thus do not impose limitations regarding the size of practically relevant multidrone missions.

[19]https://github.com/google/pprof

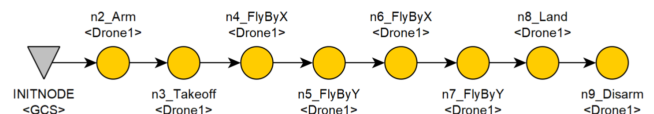

FIGURE 15. Experiment A: Mission Execution Graph and Mission Dependency Graph for the EAMOS mission of Drone1 (yellow). Since this is a single drone mission, MEG and MDG are identical. The graph is compiled from the specification of Listing 10.

IX. DISCUSSION

This section discusses our experiments compared to related approaches as presented in Section II. First of all, it should be noted that a direct, qualitative comparison with the related approaches is hardly feasible due to the significant deviations between some of the approaches and the lack of availability of functional software. Most of the investigated text-based approaches use the declarative paradigm (mostly XML dialects) for mission specification, which does not support the specification of sequential or parallel orders of actions. This is in contrast to the imperative mission paradigm used by EAMOS.

Since EAMOS is also text-based, we focus in the following discussion only on the text-based approaches "TML", "MDL", "Perdomo", "AVCL", "Bagnitckii", "DRESS-ML" and "Director Tools" as presented in Table 2.

- **Action Synchronization**: As stated in Section II-B, none of the investigated approaches offer an explicit action synchronization as EAMOS does. However,

```
func SEQ_Experiment_A() {
  Drone1.Arm()
  Drone1.Takeoff(1.5)
  Drone1.FlyByX(1) // fly 1 m in X.
  Drone1.FlyByY(1) // fly 1 m in Y.
  Drone1.FlyByX(-1) // fly 1 m in -X.
  Drone1.FlyByY(-1) // fly 1 m in -Y.
  Drone1.Land()
  Drone1.Disarm()}
```

LISTING 10. Experiment A: mission specification. After arming, the single drone takes off, flies along a rectangular path, lands at its starting point, and disarms. Pause-actions that were originally placed between actions were removed to favor readability.

```
func SEQ_Experiment_B() {
  Drone1.Arm()
  Drone2.Arm()
  PAR_Takeoff()
  PAR_Fly1()
  PAR_Fly2()
  PAR_Fly3()
  PAR_Fly4()
  Drone1.FlyByZ(0.5)
  Drone2.FlyByZ(0.5)
  PAR_Land()
  PAR_Disarm()}

func PAR_Takeoff() {
  Drone1.Takeoff(1.5)
  Drone2.Takeoff(1.5)}

func PAR_Fly1() {
  Drone1.FlyByX(1)
  Drone2.FlyByX(1)}

func PAR_Fly2() {
  Drone1.FlyByY(1)
  Drone2.FlyByY(1)}

func PAR_Fly3() {
  Drone1.FlyByX(-1)
  Drone2.FlyByX(-1)}

func PAR_Fly4() {
  Drone1.FlyByY(-1)
  Drone2.FlyByY(-1)}

func PAR_Land() {
  Drone1.Land()
  Drone2.Land()}

func PAR_Disarm() {
  Drone1.Disarm()
  Drone2.Disarm()}
```

LISTING 11. Experiment B: mission specification. After arming sequentially, both drones take off in parallel and fly along a rectangular path. The drones fly along one side of the rectangle simultaneously and wait until both reach the corresponding corner point before continuing with the next side. Pause-actions that were originally placed between actions were removed to favor readability.

```
func SEQ_Experiment_C()
  PAR_ARM()
  Drone3.Takeoff(5)
  PAR_Takeoff()
  Drone3.FlyByX(5)
  PAR_Fly1()
  Drone3.FlyByY(5)
  PAR_Fly2()
  Drone3.FlyByX(-5)
  PAR_Fly3()
  Drone3.FlyByY(-5)
  PAR_Fly4()
  Drone3.Land()
  PAR_Land()
  PAR_Disarm()}

func PAR_ARM() {
  Drone1.Arm()
  Drone2.Arm()
  Drone3.Arm()}

func PAR_Takeoff() {
  Drone1.Takeoff(1.5)
  Drone2.Takeoff(1.5)}

func PAR_Fly1() {
  Drone1.FlyByX(1)
  Drone2.FlyByX(1)}

func PAR_Fly2() {
  Drone1.FlyByY(1)
  Drone2.FlyByY(1)}

func PAR_Fly3() {
  Drone1.FlyByX(-1)
  Drone2.FlyByX(-1)}

func PAR_Fly4() {
  Drone1.FlyByY(-1)
  Drone2.FlyByY(-1)}

func PAR_Land() {
  Drone1.Land()
  Drone2.Land()}

func PAR_Disarm() {
  Drone1.Disarm()
  Drone2.Disarm()
  Drone3.Disarm()}
```

LISTING 12. Experiment C: mission specification. After taking off, the real drones, Drone1 and Drone2, make four movements along a rectangular path. At the same time, Drone3 is also flying along a rectangular path within its virtual world. By using the parallel blocks within the sequential block, the multidrone mission synchronizes the movements of the real drones with the ones of the virtual drone, such that the real drones do not make their moves before Drone3 has finished its move.

"TML" provides event handlers and event tags, which might be used for synchronizing events, but these are not intended to synchronize actions on such a detailed level as we did in our experiments (e.g., drones wait for each other until they reach a particular point). Although "MDL" supports multiple drones, there seems to be no way to control the execution order of drone actions. MDL's team description language and mission description language define groups of drones and assign tasks (e.g., extinguishing a forest fire). Still, individual drone synchronization is out of the scope of "MDL". "Director Tools" behave similarly by defining camera shooting objectives, which result in a mission specification that is fed into a mission planner, which then determines how many drones are required to perform the mission and how the drones fly without involving the user in the drone's synchronization. The other approaches do not support multiple drones.

- **Specification Syntax**: "TML", "MDL", "Perdomo", "AVCL" and "Director Tools" use XML dialects, which make their mission specifications syntactically lengthy and complex to write without a supporting syntax editor. Using any of these notations, the length

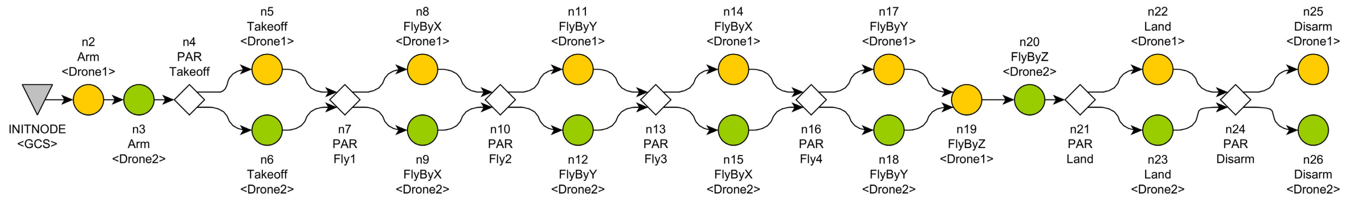

FIGURE 16. Experiment B: Mission Execution Graph for Drone1 (yellow) and Drone2 (green). Nodes not yet assigned to a drone are kept in white. The graph is compiled from the specification of Listing 11.

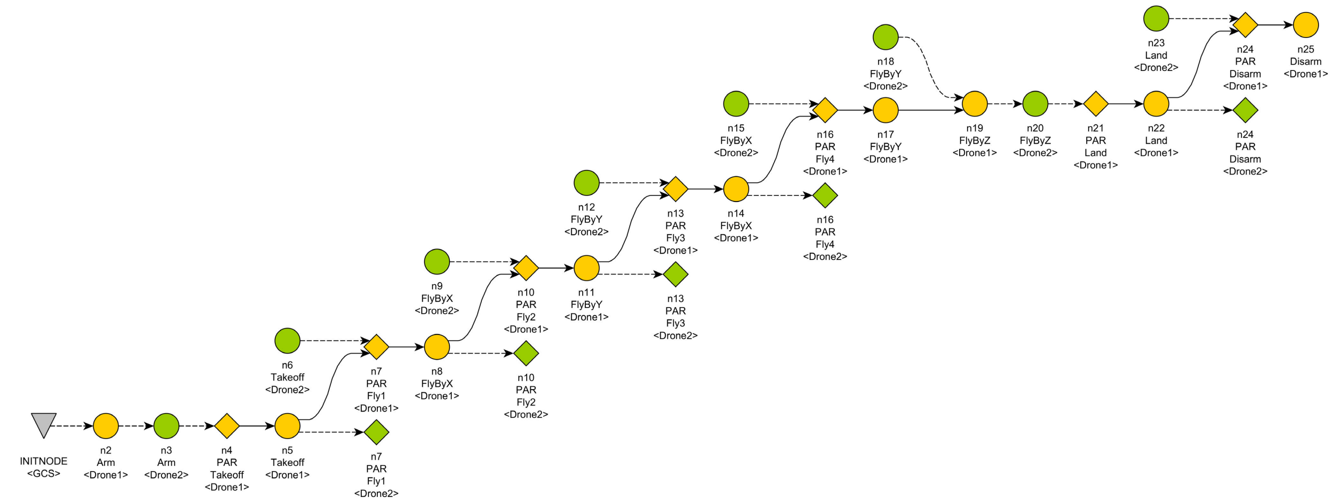

FIGURE 17. Experiment B: Mission Dependency Graph of Drone1 (yellow) indicating the dependencies to Drone2 (green). Solid lines represent local dependencies, while dashed lines represent external dependencies. The graph is compiled from the specification of Listing 11.

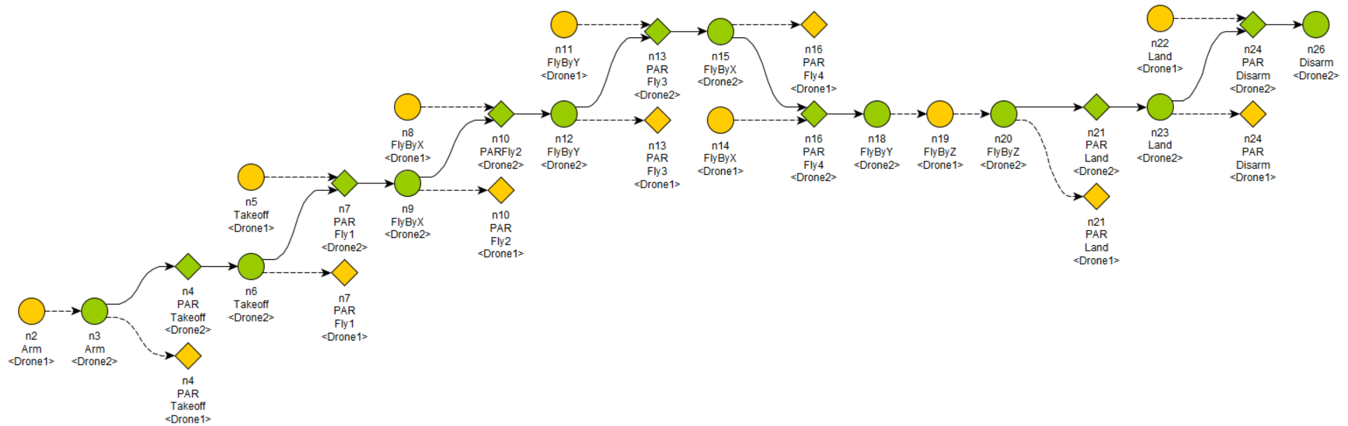

FIGURE 18. Experiment B: Mission Dependency Graph of Drone2 (green) indicating the dependencies to Drone1 (yellow). Solid lines represent local dependencies, while dashed lines represent external dependencies. The graph is compiled from the specification of Listing 11.

of the missions of our experiments would significantly increase, and their readability would suffer. Although the "Director Tools" GUI helps to increase usability, it is so specialized that it is not applicable to domains other than media productions. In contrast, "Bagnitckii" clearly supports the usability of its missions and offers a clean and concise notation that allows one to focus on the mission content rather than on the syntax of the mission. The special Given-When-Then notation of "DRESS-ML" favors the readability of their mission fragments, aiming for good mission editing usability. Nevertheless, this notation is tailored to exceptional situations rather than regular mission flows, making it barely applicable to standard scenarios as in our experiments A–D.

- **Architecture and Platforms**: "TML" seems to be based on a central execution architecture. However, since it uses the generic Aerostack[3] framework, it can

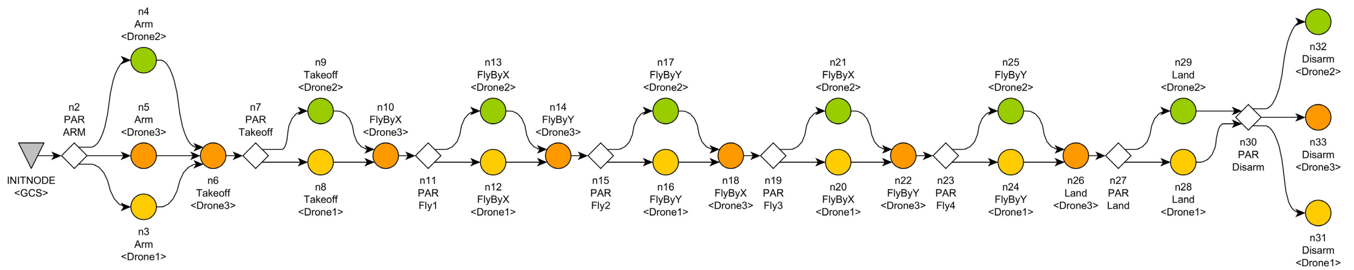

FIGURE 19. Experiment C: Mission Execution Graph for Drone1 (yellow), Drone2 (green) and Drone3 (orange). Nodes not yet assigned to a drone are kept in White. The graph is compiled from the specification of Listing 12.

be used with heterogeneous drone platforms. All other approaches use a centralized architecture or do not indicate a distributed mission execution. "AVCL", "Perdomo" and "Bagnitckii" target underwater vehicles only, which makes it hard to compare them with our experiments and the aspect of drone platform interoperability. While "DRESS-ML" does not provide any information on compatible platforms, the "Director Tools" only describe a single drone platform in the context of their work. None of the discussed text-based approaches mention ROS in any context.

- **Positioning**: As EAMOS, "TML", "Perdomo", "AVCL" and "Director Tools" allow using absolute coordinates for specifying locations, while the other approaches use geographic coordinates in the WGS84-format (latitude, longitude, altitude), which affects the resolution of the activity space of the drones and makes it rather inconvenient to use locations on a centimeter or meter scale.
- **Virtual Drones**: "MDL" and "Bagnitckii" have neither been tested in simulations or real experiments. All other approaches have been tested in at least simulations. However, none of the approaches indicate that an effortless switch between a simulated and a real environment is supported. In particular, no approach indicates that hybrid missions, such as those presented in experiments C and D, are supported or envisioned. The ability to incorporate virtual drones would open interesting possibilities regarding test and research simulations wrt. using digital twins (cf., [47]).

Overall, we draw the following conclusions from the experiments wrt. the investigated approaches.

- While EAMOS' syntax is concise, comparable mission specifications from the investigated approaches (using XML-like and similar notations) would be much more lengthy and could not be written quickly in an out-of-the-box manner.
- None of the investigated approaches provide an easy-to-read and graphical representation of the overall mission that is automatically provided by the framework, such as EAMOS provides its different graph representations that show the mission from its beginning to its end.
- Due to the fully decentralized architecture in EAMOS, we observed during our experiments that if any drone fails to continue for whatever reason, the remaining drones not dependent on the failing drone are unaffected and could continue with their part of the multidrone mission. For the few investigated approaches that offer a decentralized execution, it is not always clear how such problems would affect the overall mission.
- To the best of our knowledge, we assume that experiments B, C, and D could not be realized with any of the investigated approaches due to the described reasons.

X. CONCLUSION

To our knowledge, EAMOS[20] is the first multidrone mission-execution framework facilitating an intuitive mission specification with advanced control structures and supporting transparent mission execution in a simulation environment, on real ROS-based drone platforms, and in a hybrid setting. With EAMOS, we have developed a complete mission execution framework consisting of a compiler that processes text-based multidrone missions, an execution engine that runs onboard drone platforms and executes drone missions, as well as a middle layer tier that sits on top of ROS and enables the incorporation of different drone platforms.

We successfully showed the feasibility and correctness of our approach by conducting four different experiments with real and virtual drones, each demonstrating different aspects and capabilities of our framework. Through the experiments, we demonstrated that the size of the resulting mission graphs (in terms of the number of nodes and arcs) was reasonably small and in accordance with the input mission specifications. Moreover, we showed that the numbers of internal and external arcs of the statically generated MDGs exactly corresponded to the number of received and sent ROS service calls. All mission actions were precisely executed as defined in the mission specification. The mission execution times were almost constant for repeated experiments, with only slight variations due to the physical drone movement. While conducting the experiments, compiling and deploying multidrone missions was reasonably short for performing

[20]The entire source code of EAMOS is available as a research prototype on GitHub. Contact the first author to get access.

```
1  func SEQ_Experiment_D() {
2    D1.Arm()
3    D2.Arm()
4    D3.Arm()
5    PAR_1()
6    SEQ_2()
7    PAR_Loop()}
8
9  func PAR_1() {
10   D1.Arm()
11   D2.Arm()
12   D3.Arm()}
13
14 func SEQ_2() {
15   D1.Takeoff(1)
16   D2.Takeoff(1)
17   D3.Takeoff(1)}
18
19 func PAR_Loop() {
20   RepeatWhile(
21       D3.X() > -30 && D3.X() < 30 &&
22       D3.Y() > -30 && D3.Y() < 30,
23         SEQ_IsD1XUp,PAR_Exit)
24
25   RepeatWhile(
26       D3.X() > -30 && D3.X() < 30 &&
27       D3.Y() > -30 && D3.Y() < 30,
28         SEQ_IsD1XDown,PAR_Exit)
29
30   RepeatWhile(
31       D3.X() > -30 && D3.X() < 30 &&
32       D3.Y() > -30 && D3.Y() < 30,
33         SEQ_IsD2XUp,PAR_Exit)
34
35   RepeatWhile(
36       D3.X() > -30 && D3.X() < 30 &&
37       D3.Y() > -30 && D3.Y() < 30,
38         SEQ_IsD2XDown,PAR_Exit)}
39
40 func SEQ_IsD1XUp() {
41   WaitUntil(
42       D3.X() > -20 && D3.X() < -10 &&
43       D3.Y() > -15 && D3.Y() < -5,
44         SEQ_upXD1)}
45
46 func SEQ_IsD1XDown() {
47   WaitUntil(
48       D3.X() < -10 && D3.X() > -20 &&
49       D3.Y() > 5 && D3.Y() < 10,
50         SEQ_downXD1)}
51
52 func SEQ_IsD2XUp() {
53   WaitUntil(
54       D3.X() < 20 && D3.X() > 10 &&
55       D3.Y() < -5 && D3.Y() > -15,
56         SEQ_upXD2)}
57
58 func SEQ_IsD2XDown() {
59   WaitUntil(
60       D3.X() > 10 && D3.X() < 20 &&
61       D3.Y() < 10 && D3.Y() > 5,
62         SEQ_downXD2)}
63
64 func SEQ_upXD1() {
65   D1.FlyByX(0.2)}
66
67 func SEQ_downXD1() {
68   D1.FlyByX(-0.2)}
69
70 func SEQ_upXD2() {
71   D2.FlyByX(0.2)}
72
73 func SEQ_downXD2() {
74   D2.FlyByX(-0.2)}
75
76 func PAR_Exit() {
77   D3.Land()}
```

LISTING 13. Experiment D: mission specification. The real drones Drone1 and Drone2 take off and move only when the virtual Drone3 is above one of four quadrants within its virtual world, which are named as follows: upXD1–let Drone1 move forward; downXD1–let Drone1 move backward; upXD2–let Drone2 move forward; downXD2–let Drone2 move backward. The mission control structure RepeatWhile is used to loop the mission logic as long as Drone3 is above the overall test area (defined by the rectangle (-30/-30/0) to (30/30/0) that includes the four mentioned quadrants). Another mission control structure, WaitUntil, is then used to issue the movement of a real drone whenever Drone3 is above one of the quadrants (also defined by their rectangles).

re-compilations and continuous deployments in an ongoing workflow. Finally, our decision to implement EAMOS from scratch was based on its key capabilities, such as explicit synchronization, distributed execution engine, platform versatility, and the limited software availability of related approaches.

A. REAL-WORLD APPLICATIONS

Though this work has focused on fundamental concepts and prototypical implementation of mission specification and execution, we briefly discuss implementation aspects of potential real-world applications with EAMOS. First, a strength of our framework from an end user's perspective is the ability to aggregate basic actions into special activities through reusable functions. Since EAMOS mission specifications are Go source code, mission specifications can be imported by other mission specifications to use their functions, similar to external program libraries for a computer program. Second, drone platforms provide a steadily increasing set of hardware and software capabilities that can be integrated via the EAMOS Middle Layer. Naturally, the actions provided by the public API strongly influence the scope of potential applications.

We foresee several real-world applications that can be specified and executed with EAMOS. Monitoring and inspection applications often require multiple drones with specific actions that must be synchronized. Coordination and synchronization are typically required at a higher level and triggered by conditions achieved from the actions. Search and surveillance missions are another application domain with specific (search) actions for multiple drones and condition-based synchronization. Regardless of the specific application, a real-world deployment will definitely require implementation effort for additional EAMOS functionality and the integration of drone capabilities into the middle layer.

B. LIMITATIONS

To round up the discussion, we enlist some limitations of the current EAMOS implementation and reflect on dynamic drone environments and mission deviations.

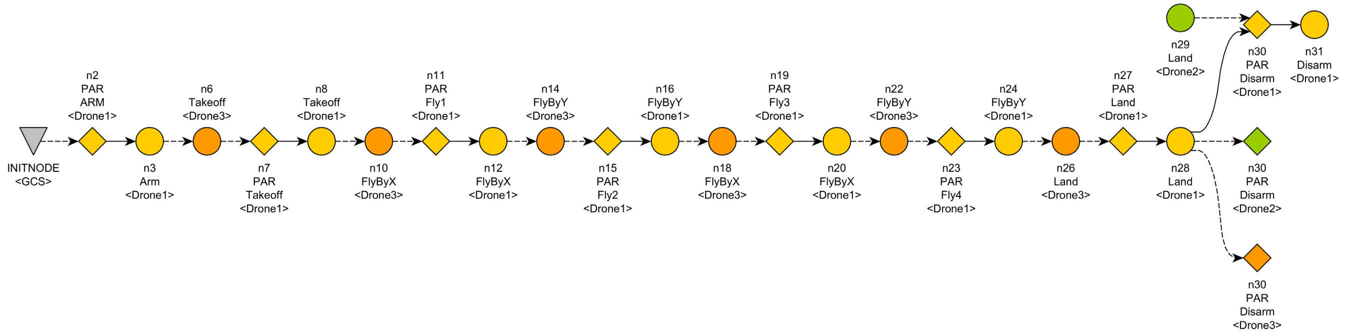

FIGURE 20. Experiment C: Mission Dependency Graph of Drone1 (yellow) indicating the dependencies to Drone2 (green) and Drone3 (orange). Solid lines represent local dependencies, while dashed lines represent external dependencies. The graph is compiled from the specification of Listing 12.

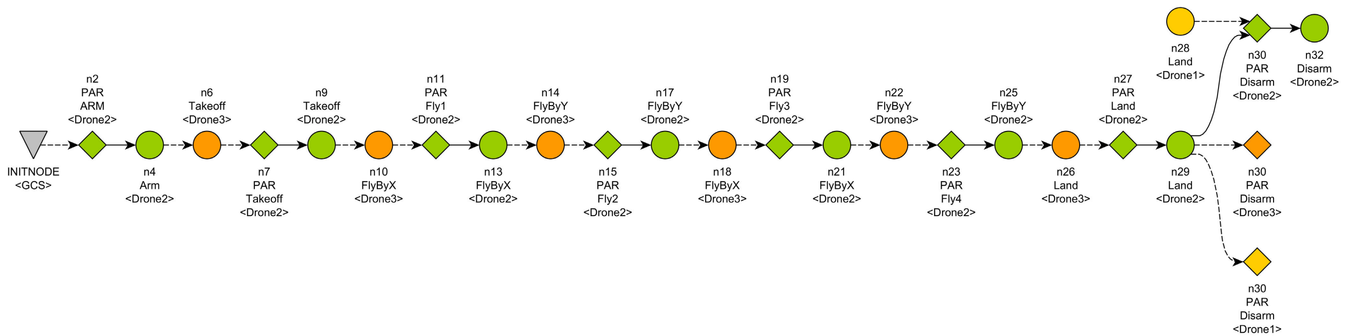

FIGURE 21. Experiment C: Mission Dependency Graph of Drone2 (green) indicating the dependencies to Drone1 (yellow) and Drone3 (orange). Solid lines represent local dependencies, while dashed lines represent external dependencies. The graph is compiled from the specification of Listing 12.

1) EAMOS has been developed as a research prototype with a small set of implemented actions that naturally limit the functionality and complexity of missions. However, additional actions can be easily integrated via EAMOS' Middle Layer.
2) EAMOS does currently not support adaptations of missions during execution. EAMOS compiles and deploys complete missions prior to execution and executes them until they terminate or are shut down forcefully.
3) For increased flexibility and portability, EAMOS is built upon several external components and libraries, such as MAVProxy, MAVROS, and ROS' action_lib packages. EAMOS' functionality depends on the availability and evolution of these external entities.
4) Although EAMOS is designed as a decentralized framework, the current implementation based on ROS version 1 requires a dedicated primary node, which puts one drone (or, optionally, a base station) in charge of maintaining the ROS network. An upgrade to ROS version 2 would remove this limitation.

In its current state, EAMOS deals with uncertainties in the drone's environment through its palette of mission control structures (cf., Section VI-D). Loops and conditions, if used in concurrent execution branches of a mission (), easily resemble the functionality of event handlers, which are running in the background and respond to external events. This was demonstrated by experiment D (cf., Section VIII-D), in which the RepeatWhile and WaitUntil structures were used to test whether the virtual drone was within a particular region, what was then responded by the real drones.

Partial deployment of the multidrone mission can handle mission deviations that might occur during the runtime of a mission. This is possible because EAMOS supports replacing drone mission programs onbord drones while the overall mission executes. This means that if a mission changes, the mission specification can be adapted and newly compiled, and then only those drone mission programs that contain changed mission parts might be deployed to the drones.

C. FUTURE WORK

Overall, we envision promising potential in EAMOS and see several examples for further advancing our framework.

1) **Advanced Actions**: A natural next step is to add more powerful actions to the Uniform API and implement them for one or more drone platforms. Examples of actions include handling onboard cameras (e.g., data capturing, processing, and data delivery) or realizing advanced flight maneuvers (e.g., orbiting and specific flight patterns).
2) **Action Interruption**: Currently, ROS' action_lib lets us use custom execution feedback and the ability to send a cancellation request from an action client to an action server. However, adapter actions can

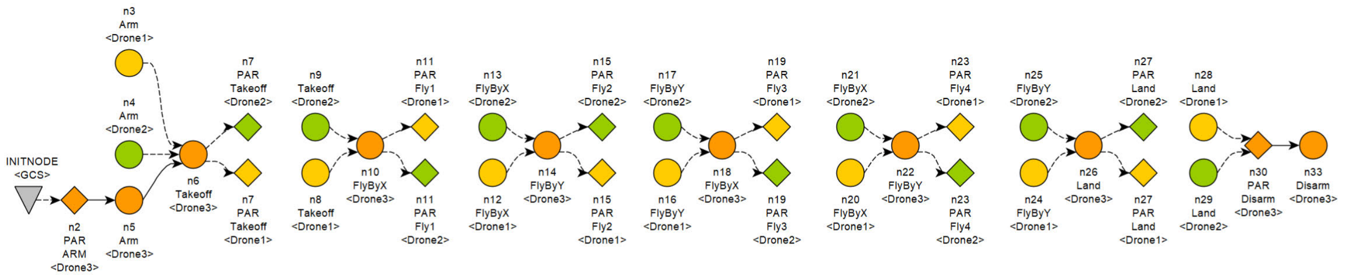

FIGURE 22. Experiment C: Mission Dependency Graph of Drone3 (orange) indicating the dependencies to Drone1 (yellow) and Drone2 (green). Solid lines represent local dependencies, while dashed lines represent external dependencies. The graph is compiled from the specification of Listing 12.

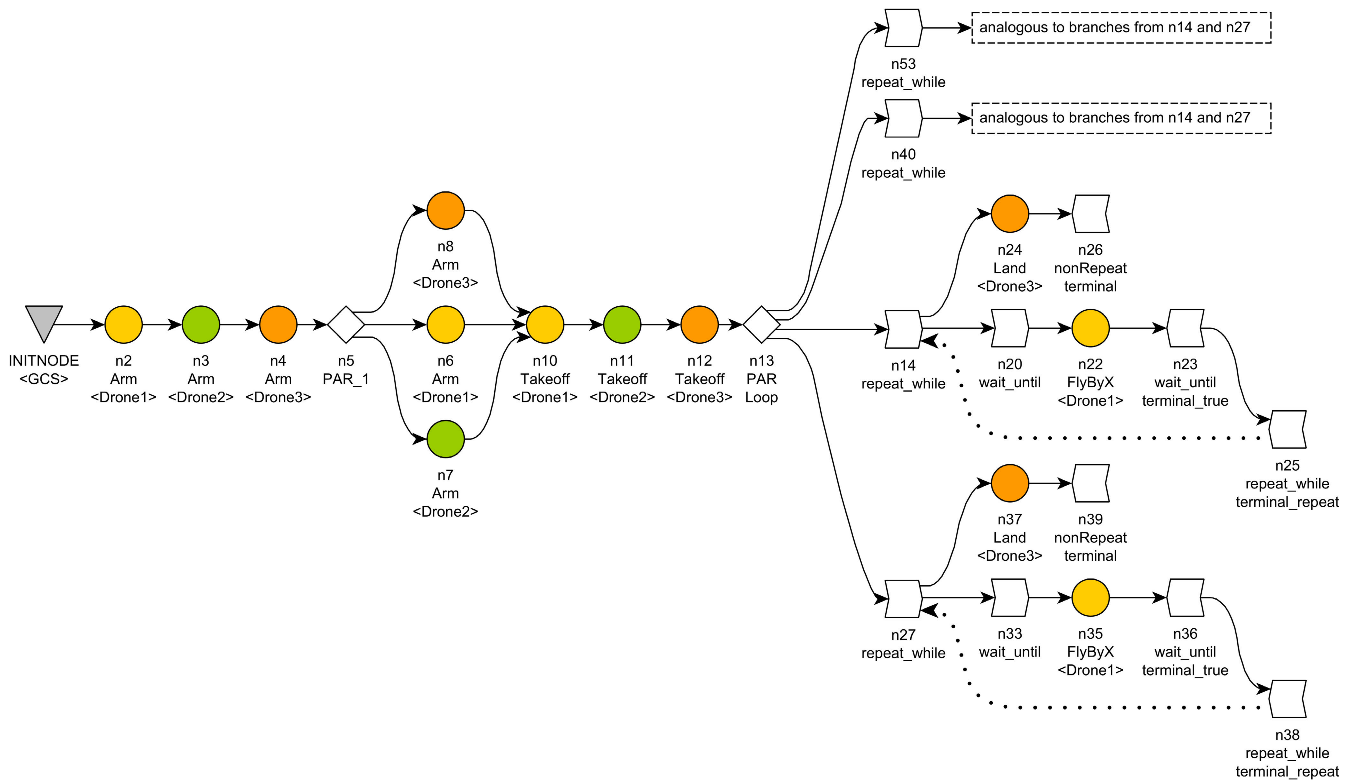

FIGURE 23. Experiment D: Mission Execution Graph for Drone1 (yellow), Drone2 (green), and Drone3 (orange). Nodes not yet assigned to a drone are kept in white. Right-arrow nodes indicate condition beginnings (repeat_while and wait_until), and left-arrow nodes indicate terminations of conditions (e.g., terminal_repeat). Dotted lines indicate loops. The graph is compiled from the specification of Listing 13.

receive cancellation requests, but since each adapter action requires its individual cancellation handling to guarantee sound action control, the cancellation mechanisms are not yet implemented.

3) **Event Handlers**: An introduction of real event handlers that replace the "busy waiting"-logic of the mission control logic shown in Experiment D that were used to react on external events will support usability and improve runtime performance.
4) **Exception Handling**: The integration of mission exception handling would improve the reliability of EAMOS mission execution. This enables mission authors to specify actions in case of unexpected events, improving the robustness and safety of executing missions.
5) **Dynamic Mission Adaptations**: Changes to a running mission are envisioned to be supported by EAMOS on the mission action level rather than on the mission graph level. This means that the drone execution environment shall be able to adapt individual MDGs of executing drone mission programs according to requests to the program from outside.
6) **Source Code Injection**: Integrating arbitrary Go code into mission specifications would improve flexibility for expert mission authors. The injected code must be transparently compiled and deployed on the drone's intermediate mission file.
7) **User Interface**: Since EAMOS missions conform with Go syntax, writing them is already convenient using the

integrated development environment (IDE). However, a dedicated EAMOS IDE with syntax highlighting, code completion, and mission monitoring features would further enhance the user experience of mission authors.

To conclude, we are confident that EAMOS will help to simplify mission specification and automate mission execution, enabling novel multidrone applications.

ACKNOWLEDGMENT

The authors would like to thank Hermann Zunter for his assistance with the drone software and Rockson Agyeman, Kyriakos Lite, and Vitali Korzhun for their support with the drone experiments.

REFERENCES

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

[2] D. C. Tsouros, S. Bibi, and P. G. Sarigiannidis, "A review on UAV-based applications for precision agriculture," *Information*, vol. 10, no. 11, p. 349, Nov. 2019.

[3] S. Guan, Z. Zhu, and G. Wang, "A review on UAV-based remote sensing technologies for construction and civil applications," *Drones*, vol. 6, no. 5, p. 117, May 2022.

[4] S. Jordan, J. Moore, S. Hovet, J. Box, J. Perry, K. Kirsche, D. Lewis, and Z. T. H. Tse, "State-of-the-art technologies for UAV inspections," *IET Radar, Sonar Navigat.*, vol. 12, no. 2, pp. 151–164, Feb. 2018.

[5] D. Sziroczak, D. Rohacs, and J. Rohacs, "Review of using small UAV based meteorological measurements for road weather management," *Prog. Aerosp. Sci.*, vol. 134, Oct. 2022, Art. no. 100859.

[6] N. Dilshad, J. Hwang, J. Song, and N. Sung, "Applications and challenges in video surveillance via drone: A brief survey," in *Proc. Int. Conf. Inf. Commun. Technol. Converg. (ICTC)*, Oct. 2020, pp. 728–732.

[7] V. C. Hollman, "Drone photography and the re-aestheticisation of nature," in *Decolonising and Internationalising Geography*. Cham, Switzerland: Springer, 2020, pp. 57–66. [Online]. Available: https://link.springer.com/book/10.1007/978-3-030-49516-9#bibliographic-information

[8] I. Karakostas, I. Mademlis, N. Nikolaidis, and I. Pitas, "Shot type feasibility in autonomous UAV cinematography," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2019, pp. 1937–1941.

[9] M. De Marsico and A. Spagnoli, "Using hands as an easy UAV joystick for entertainment applications," in *Proc. 13th Biannual Conf. Italian SIGCHI Chapter, Designing Next Interact.*, Sep. 2019, pp. 1–9.

[10] J. Scherer, S. Yahyanejad, S. Hayat, E. Yanmaz, T. Andre, A. Khan, V. Vukadinovic, C. Bettstetter, H. Hellwagner, and B. Rinner, "An autonomous multi-UAV system for search and rescue," in *Proc. 1st Workshop Micro Aerial Vehicle Netw., Syst., Appl. Civilian Use*, May 2015, pp. 33–38.

[11] E. Allak, C. Brommer, D. Dallenbach, and S. Weiss, "AMADEE-18: Vision-based unmanned aerial vehicle navigation for analog Mars mission (AVI-NAV)," *Astrobiology*, vol. 20, no. 11, pp. 1321–1337, Nov. 2020.

[12] M. Quaritsch, R. Kuschnig, H. Hellwagner, and B. Rinner, "Fast aerial image acquisition and mosaicking for emergency response operations by collaborative UAVs," in *Proc. Int. Conf. Inf. Syst. Crisis Response Manage.*, 2011, pp. 1–5.

[13] P. Mazdin, K. P. Kolleg, and B. Rinner, "Efficient and QoS-aware drone coordination for simultaneous environment coverage," in *Proc. IEEE Conf. Multimedia Inf. Process. Retr. (MIPR)*, Mar. 2019, pp. 333–338.

[14] E. Vrochidou, V. N. Tsakalidou, I. Kalathas, T. Gkrimpizis, T. Pachidis, and V. G. Kaburlasos, "An overview of end effectors in agricultural robotic harvesting systems," *Agriculture*, vol. 12, no. 8, p. 1240, Aug. 2022.

[15] T. Benarbia and K. Kyamakya, "A literature review of drone-based package delivery logistics systems and their implementation feasibility," *Sustainability*, vol. 14, no. 1, p. 360, Dec. 2021.

[16] G. Gugan and A. Haque, "Path planning for autonomous drones: Challenges and future directions," *Drones*, vol. 7, no. 3, p. 169, Feb. 2023.

[17] M.-T.-O. Hoang, N. van Berkel, M. B. Skov, and T. R. Merritt, "Challenges and requirements in multi-drone interfaces," in *Proc. Extended Abstr. CHI Conf. Hum. Factors Comput. Syst.*, Apr. 2023, pp. 1–9.

[18] W. Chen, J. Liu, H. Guo, and N. Kato, "Toward robust and intelligent drone swarm: Challenges and future directions," *IEEE Netw.*, vol. 34, no. 4, pp. 278–283, Jul. 2020.

[19] M. Gutmann and B. Rinner, "Mission specification and execution of multidrone systems," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Feb. 2021, pp. 451–456.

[20] M. Gutmann and B. Rinner, "EAMOS: Execution of aerial multidrone mission operations and specifications framework," in *Proc. Int. Conf. Unmanned Aircr. Syst. (ICUAS)*, Jun. 2023, pp. 761–768.

[21] 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. Int. Micro Air Vehicle Conf. Competition (IMAV)*, 2016, pp. 1–8.

[22] M. Molina, R. A. Suarez-Fernandez, C. Sampedro, J. L. Sanchez-Lopez, and P. Campoy, "TML: A language to specify aerial robotic missions for the framework aerostack," *Int. J. Intell. Comput. Cybern.*, vol. 10, no. 4, pp. 491–512, Nov. 2017.

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

[24] D. C. Silva, P. H. Abreu, L. P. Reis, and E. Oliveira, "Development of a flexible language for disturbance description for multi-robot missions," *J. Simul.*, vol. 10, no. 3, pp. 166–181, Aug. 2016.

[25] D. C. Silva, P. H. Abreu, L. P. Reis, and E. Oliveira, "Development of flexible languages for scenario and team description in multirobot missions," *Artif. Intell. Eng. Design, Anal. Manuf.*, vol. 31, no. 1, pp. 69–86, Feb. 2017.

[26] E. F. Perdomo, J. C. Gámez, A. C. D. Brito, and D. H. Sosa, "Mission specification in underwater robotics," *J. Phys. Agents*, vol. 4, no. 1, pp. 25–33, Jan. 2010.

[27] D. Davis, "Automated parsing and conversion of vehicle-specific data into autonomous vehicle control language (AVCL) using context-free grammars and XML data binding," in *Proc. 14th Int. Symp. Unmanned Untethered Submersible Technol.*, 2005, pp. 1–11.

[28] D. Davis and D. Brutzman, "The autonomous unmanned vehicle workbench: Mission planning, mission rehearsal, and mission replay tool for physics-based X3D visualization," in *Proc. 14th Int. Symp. Unmanned Untethered Submersible Technol.*, 2005, pp. 1–12.

[29] A. Bagnitckii, A. Inzartsev, and R. Senin, "Facilities of AUV search missions planning," in *Proc. OCEANS MTS/IEEE KONA*, Sep. 2011, pp. 1–7.

[30] 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 Int. Conf. Softw. Eng., Softw. Eng. Soc. (ICSE-SEIS)*, May 2022, pp. 56–66.

[31] A. Torres-González, A. Alcántara, V. Sampaio, J. Capitán, B. Guerreiro, R. Cunha, and A. Ollero, "Distributed mission execution for aerial cinematography with multiple drones," in *Proc. Workshop Signal Process. Comput. Vis. Deep Learn. Auto. Syst.*, 2019, pp. 1–5.

[32] Á. Montes-Romero, A. Torres-González, J. Capitán, M. Montagnuolo, S. Metta, F. Negro, A. Messina, and A. Ollero, "Director tools for autonomous media production with a team of drones," *Appl. Sci.*, vol. 10, no. 4, p. 1494, Feb. 2020.

[33] N. Paula, B. Areias, A. B. Reis, and S. Sargento, "Multi-drone control with autonomous mission support," in *Proc. IEEE Int. Conf. Pervasive Comput. Commun. Workshops (PerCom Workshops)*, Mar. 2019, pp. 918–923.

[34] 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 *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 801–806.

[35] F. Ciccozzi, D. D. Ruscio, I. Malavolta, and P. Pelliccione, "Adopting MDE for specifying and executing civilian missions of mobile multi-robot systems," *IEEE Access*, vol. 4, pp. 6451–6466, 2016.

[36] D. Di Ruscio, I. Malavolta, and P. Pelliccione, "Engineering a platform for mission planning of autonomous and resilient quadrotors," in *Software Engineering for Resilient Systems*. Berlin, Germany: Springer, 2013, pp. 33–47. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-40894-6_3#citeas

[37] D. Ruscio, I. Malavolta, and P. Pelliccione, "A family of domain-specific languages for specifying civilian missions of multi-robot systems," in *Proc. 1st Int. Workshop Model-Driven Robot Softw. Eng. (MORSE)*, 2014, pp. 13–26.

[38] J. A. Besada, A. M. Bernardos, L. Bergesio, D. Vaquero, I. Campaña, and J. R. Casar, "Drones-as-a-service: A management architecture to provide mission planning, resource brokerage and operation support for fleets of drones," in *Proc. IEEE Int. Conf. Pervasive Comput. Commun. Workshops (PerCom Workshops)*, Mar. 2019, pp. 931–936.

[39] A. P. Lamping, J. N. Ouwerkerk, and K. Cohen, "Multi-UAV control and supervision with ROS," in *Proc. Aviation Technol., Integr., Oper. Conf.*, Jun. 2018, p. 4245.

[40] S. Dragule, T. Berger, C. Menghi, and P. Pelliccione, "A survey on the design space of end-user-oriented languages for specifying robotic missions," *Softw. Syst. Model.*, vol. 20, no. 4, pp. 1123–1158, Aug. 2021.

[41] S. Magnenat, V. Longchamp, and F. Mondada, "ASEBA, An event-based middleware for distributed robot control," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (Workshops Tutorials)*, Nov. 2007, pp. 1–6.

[42] S. García, P. Pelliccione, C. Menghi, T. Berger, and T. Bures, "PROMISE: High-level mission specification for multiple robots," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng., Companion Proc. (ICSE-Companion)*, Oct. 2020, pp. 5–8.

[43] C. A. R. Hoare, *Communicating Sequential Processes*. Upper Saddle River, NJ, USA: Prentice-Hall, 1985.

[44] C. A. R. Hoare and A. W. Roscoe, *The Laws of OCCAM Programming*. Oxford, U.K.: Oxford Univ. Computing Laboratory, 1986.

[45] C. A. Petri, *Kommunikation mit Automaten*. Bonn, Germany: Rheinisch-Westfälisches Institut für Instrumentelle Mathematik an der Universität, 1962.

[46] K. Jensen, "A brief introduction to coloured Petri nets," in *Proc. Int. Workshop Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer, 1997, pp. 203–208. [Online]. Available: https://link.springer.com/chapter/10.1007/BFb0035389

[47] T. Souanef, S. Al-Rubaye, A. Tsourdos, S. Ayo, and D. Panagiotakopoulos, "Digital twin development for the airspace of the future," *Drones*, vol. 7, no. 7, p. 484, Jul. 2023.

MARKUS GUTMANN received the B.Sc. and M.Sc. degrees in applied informatics from the University of Klagenfurt, in 2012 and 2015, respectively, where he is currently pursuing the Ph.D. degree with the Institute of Networked and Embedded Systems. He is a Research Assistant with the Institute of Networked and Embedded Systems, University of Klagenfurt. He was formerly with the Transportation Informatics Research Group, University of Klagenfurt, focusing on software development for vehicle simulations. He has several years of experience as a professional software developer. Since 2015, he has been a "Confirmed Software Engineer" in a company specializing in factory automation, where he develops solutions in the context of manufacturing processes for semiconductors, focusing on test automation and software quality assurance. His research interests include exploiting usability, drone synchronization, and platform independence for multidrone systems and their missions.

BERNHARD RINNER (Senior Member, IEEE) received the M.Sc. and Ph.D. degrees in telematics from the Graz University of Technology, Graz, Austria, in 1993 and 1996, respectively. He held research positions with the Graz University of Technology, from 1993 to 2007, and The University of Texas at Austin, Austin, TX, USA, from 1998 to 1999. He is currently a Professor of pervasive computing and the Vice Dean of the Faculty of Technical Sciences, University of Klagenfurt, Austria. He has authored and coauthored more than 250 articles for journals, conferences, and workshops, and has led many research projects. His current research interests include sensor networks, multirobot systems, self-organization, and pervasive computing. He has served as a reviewer, a program committee member, the program chair, and the editor-in-chief. He is currently an Associate Editor of the *Ad Hoc Networks* journal and a member of the board of the Austrian Science Fund.