2018 International Conference on Unmanned Aircraft Systems (ICUAS)
Dallas, TX, USA, June 12-15, 2018

A Brief Introduction to Unmanned Systems Autonomy Services (UxAS)

Steven Rasmussen, Derek Kingston, Laura Humphrey

***Abstract*—Future concepts for autonomy envisage teams of unmanned aerial vehicles (UAVs) performing a variety of missions quickly, efficiently, and with minimal human oversight. However, developers of mission-level autonomy for UAV teams face many challenges. These include the need to support ad hoc inter-UAV communication, to provide a baseline set of autonomous capabilities, to support the ongoing incorporation of new autonomous capabilities, and to manage the inherent complexity of the software. To address these challenges, we developed the Unmanned Systems Autonomy Services (UxAS), an extensible software framework for mission-level autonomy for teams of unmanned systems, with a focus on UAVs. UxAS was borne out of over 15 years of research and 10 years of flight testing and is now publicly available and free to use.**

We start with a discussion of mission-level autonomy, followed by an overview of UxAS. Next, we provide a high-level comparison to other software frameworks and a more in-depth comparison to the Robot Operating System. Then, we provide details on UxAS services and configuration, discuss how a core set of services can be configured to work together to plan and execute a multi-UAV surveillance mission, describe utilization and flight testing of UxAS, and discuss areas of future work.

I. Introduction

Unmanned Aerial Vehicles (UAVs) are becoming increasingly capable. However, standard methods for UAV control such as a human operator manually using a joystick or constructing waypoint-based plans are labor-intensive and not generally practical for teams of UAVs. Future concepts envisage autonomy that will not only reduce the amount of labor needed to control teams of UAVs, but allow them to quickly and efficiently perform missions too complex or dynamic to be fully pre-planned or executed online by human operators.

However, various requirements make the development of mission-level autonomy for UAV teams challenging. In order to be robust to unreliable communication channels, autonomous capabilities will need to run onboard UAVs, and UAVs will need to communicate directly with each other rather than centrally through a ground control station. UAV payload size, weight, and power (SWaP) limitations will in turn limit the available processing power of onboard computers. Software for UAV autonomy will be inherently complex, and this complexity will need to be managed. Software will also need to be easily extensible to support the ongoing incorporation of new autonomous capabilities.

We have performed research in autonomy for UAV teams for over 15 years and have been implementing this research in flight test experiments for over 10 years. In the beginning, we implemented autonomous capabilities in an ad hoc manner, leading to software that was ultimately unsupportable. Using lessons learned, we began re-architecting our software in 2013 to better meet the requirements for developing mission-level multi-UAV autonomy, with an eye toward flexibility and extensibility. The result is the Unmanned Systems Autonomy Services (UxAS), a free and publicly available software framework for mission-level autonomy for teams of unmanned systems, with an emphasis on UAVs[1]. UxAS is now being integrated into various research programs and has been successfully demonstrated in numerous flight tests.

This paper provides a brief introduction to UxAS. In Section II, we provide context by discussing what we consider to be "mission-level autonomy" for teams of unmanned systems. In Section III, we give a high-level description of UxAS, including a brief history of the development that lead to it and its resulting architecture and design rationale. In Section IV, we compare UxAS to other software frameworks for robot or multi-agent control, with an in-depth comparison to the Robot Operating System (ROS). In Section V, we discuss how to configure UxAS and its services, and in Section VI, we detail the major services comprising UxAS. In Section VII, we describe utilization and flight testing of UxAS, and in Section VIII, we discuss current and future directions.

II. Mission-Level Autonomy

In this section, we describe two missions that exemplify the major capabilities needed for mission-level autonomy for teams of unmanned systems. These capabilities include:

task planning: calculating the estimated task-level waypoints and cost required to perform a specific mission task, accounting for task options and sensor image requirements.

task assignment: assigning vehicles to tasks while minimizing mission cost and meeting possible hard constraints, e.g. on vehicle eligibility and the ordering of tasks.

task execution: controlling vehicle(s) during the task, possibly requiring inter-vehicle coordination for multi-vehicle tasks or making online updates to the original estimated task plan due to changes in mission circumstances.

path planning: calculating the *enroute* waypoints needed to move between tasks and the *on-task* waypoints needed

This paper is based on work performed at the Air Force Research Laboratory (AFRL) *Control Science Center of Excellence*. Distribution A: Approved for public release. Case #88ABW-2018-2244.

S. Rasmussen is a Principal Engineer for Miami Valley Aerospace LLC, AFRL, Wright-Patterson AFB steve.rasmussen@MVAero.com

D. Kingston is a Senior Research Engineer with the United States Air Force, AFRL, Wright-Patterson AFB derek.kingston@us.af.mil

L. Humphrey is a Research Engineer with the United States Air Force, AFRL, Wright-Patterson AFB laura.humphrey@us.af.mil

[1]https://github.com/afrl-rq/OpenUxAS

to move between task-level waypoints, accounting for keep-in zones, keep-out zones, and vehicle dynamic or kinematic constraints.

sensor management: providing the sensor information, e.g. on sensor type and sensor footprint geometry, necessary for task planning and execution.

We discuss these capabilities in the context of two example missions: basic intelligence, surveillance, and reconnaissance (ISR), and unattended ground sensor (UGS) intruder isolation. For simplicity, we restrict the discussion to UAVs. Note that these capabilities build off of an autopilot that provides guidance, navigation, and control. Also note that these capabilities must work when UAVs have only ad hoc communication.

A. Basic ISR

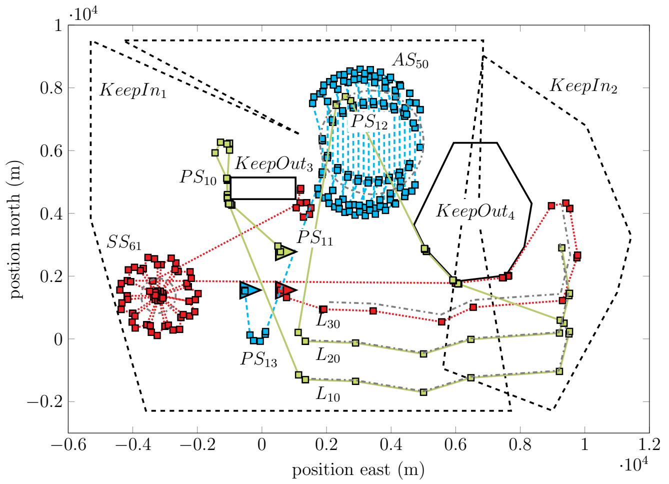

Fig. 1: A basic ISR mission. UAVs are represented as triangles.

Consider the basic ISR mission depicted in Figure 1. It includes three UAVs and four point search tasks PS_{10}, PS_{11}, PS_{12}, and PS_{13}; three line search tasks L_{10}, L_{20}, and L_{30}; an area search task AS_{50}; and a sector search task SS_{61}. For each type of task, there are options on how tasks of that type can be performed. For instance, the approach heading for each point search task can be set to any angle. Each line search task can start from one end or the other. The set of parallel passes over the area for the area search task (which are spaced according to the width of the assigned UAV's sensor footprint) can be set to any angle. The sector search task, which collects views of a specified point at many angles by making a sequence of passes that sweep through a point-centered circle of a certain radius or extent, can be set to an extent of any positive value. There are also general sensor payload options that apply to all task types. For instance, a sensor can be set to point toward the left of a UAV during a task, as is the case for L_{30}. This results in the assigned UAV's on-task path or route being offset from the line so that the sensor footprint can be centered on it during the task. Note that if nothing is specified for a particular option, any valid value can be used. See [1] for a list of options.

Suppose we want the UAVs to collectively perform all of these tasks, some with specified options, some with constraints on which UAVs are eligible to perform them, and some with constraints on their order relative to other tasks. For instance, we might specify that only the green UAV (solid path) can perform L_{10} and L_{20}, and L_{10} must be performed before L_{20}.

For each task, *task planning* calculates task-level waypoints for each combination of eligible UAV and available task option. Note that waypoints might be different for each UAV, e.g. a UAV at a higher altitude might have a larger sensor footprint, causing lanes for the area search to be spaced farther apart. Calculating the size of the sensor footprint requires *sensor management*. *Path planning* is then used to calculate dynamically or kinematically feasible paths between task-level waypoints to get the full *on-task* waypoint route and cost.

Given estimated costs from task planning, *task assignment* assigns ordered tasks to specific UAVs. The task assignment algorithm should attempt to minimize total mission cost (e.g. total distance traveled or fuel used) by searching over eligible UAVs and available task options and orderings, with *path planning* used to estimate the *enroute* paths and costs needed to fly between tasks. Note that task assignment algorithms generally use heuristics since full optimization is intractable [2]. The algorithm should also minimize inter-UAV communication. A simple minimal communication algorithm is as follows. Upon receiving a request to perform the mission, UAVs exchange information on their current state with all other UAVs that are currently in communication. After handshaking on this state information, each UAV runs the same deterministic planning and assignment algorithms. Since the state information and algorithms are the same, each UAV computes the same plan in a decentralized fashion and can then execute its assigned tasks independently. In the worst case, different groups of UAVs that are not in communication with each other after receiving the mission request will duplicate the mission. More sophisticated algorithms, e.g. consensus-based decentralized auctions, can be used to mitigate this problem [3].

In this mission, *task execution* is fairly straightforward and mainly consists steering the sensor during the task using information from *sensor management*. It could also include updating routes, e.g. if unexpected winds cause a UAV to deviate significantly from the original plan.

B. UGS Intruder Isolation

Consider a mission in which multiple UAVs must find or "isolate" a ground intruder that is moving along a road network and capture aerial imagery of it. To isolate the intruder, UAVs collect information from a set of UGSs, each of which detects and records the intruder's time, speed, and direction as it passes by. UGSs provide an alternative to automatic target recognition, which can be expensive and unreliable. UGSs are fairly cheap and can be easily placed at many points on the road network, as depicted in Figure 2. However, their communication range is limited and requires line of sight, so establishing a fully connected network of UGSs would be prohibitively expensive. Instead, UAVs fly to within communi-

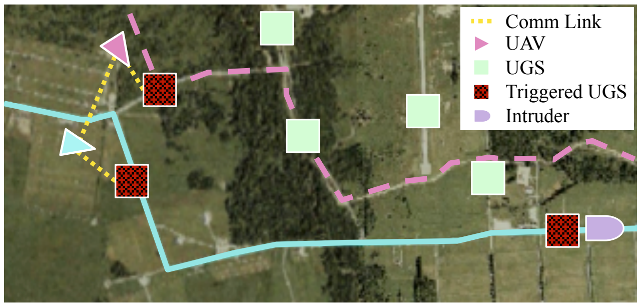

Fig. 2: The UGS intruder isolation mission.

cation range of individual UGSs to collect information. When in communication with each other, UAVs can also share UGS information and cooperate on isolating the intruder.

The algorithm for this mission, detailed in [4], is approximately as follows. UAVs share information and cooperate on isolating the intruder whenever they are in communication. When UAVs have no information on the intruder or the information is too old, they systematically visit UGSs to collect new information. When UAVs have up-to-date information on the intruder, they estimate which UGSs the intruder most likely headed to next and fly to those UGSs to collect more information. Given bounds on the intruder's speed and assuming the UAVs are faster than the intruder, the UAVs will eventually be able to determine the intruder's current road segment and next UGS location. Once a UAV has gotten close enough to the intruder, the UAV can overtake it while performing a line search of its current road segment, then wait for it by loitering at the next UGS. When the UGS is triggered, the loitering UAV will know that it has captured imagery of the intruder. If too much time elapses, e.g. because a previous UGS generated a false alarm, then the UAVs restart the mission. Note that if UAVs lose communication with each other, they can still use all available information to perform the mission individually or in smaller groups, though the odds of capturing the intruder increase as more UAVs collaborate.

This mission requires two levels of *task execution*. Top-level task execution involves collecting information from UGSs and other UAVs and making decisions about which UGSs to visit next, when to perform a line search, when to loiter at a particular UGS to capture imagery of the intruder, and when to restart the mission. In turn, this level of task execution uses *task assignment*, *task planning*, *path planning*, and *sensor management* in a manner similar to the basic ISR mission of Section II-A to determine which UAVs should collect information from specific UGSs and which UAV should perform the line search and loiter once the intruder has been isolated. Lower-level task execution involves actually performing the line search and loiter as in the basic ISR mission.

III. What is UxAS

UxAS is a software framework for mission-level autonomy for teams of unmanned systems, where the "x" serves as a placeholder for different types of systems, including unmanned air, ground, and sea surface vehicles. While UxAS has support for all of these types of systems and could easily be extended to other types of unmanned systems, its emphasis is on UAVs. We therefore discuss UxAS in the context of UAVs for the rest of the paper, though most of the same principles apply to UxAS's handling of other unmanned systems.

UxAS provides all the capabilities listed in Section II and all of the communication transport, messaging, configuration, and software control functionality required to implement UAV mission-level autonomy on single UAVs and teams of UAVs. UxAS is implemented in a service oriented architecture. That is, UxAS is designed around independent services that interact through the use of pre-defined messages. Services subscribe to messages using a publish/subscribe (pub/sub) communication pattern, which makes it easy to add new services and connect them to the rest of the system. This pub/sub communication pattern also facilitates connecting with external entities such as UGSs and other UAVs through the use of standard communication protocols. The result is a modular architecture that helps manage software complexity and enables extensibility. It is also flexible in how it is deployed; UxAS services can run in a single process, or they can run over multiple processes on one computer or many computers.

In what follows, we give a brief history of the development that led to UxAS and summarize lessons learned, then provide a description of UxAS's architecture and design rationale.

A. History

In order to give the reader a better understanding of UxAS and its design rationale, we give a brief history of its predecessors. From 1999 to 2006, we began our research in control of UAV teams, with a concentration on path planning and task assignment for simple point search tasks. The research was performed in simulation, and the software architecture was constructed in C++/MATLAB/SIMULINK [2], [5]–[7]. This architecture made it possible to simulate multiple UAVs and implement mission management algorithms. Messages were designed into Simulink connections, making it possible to send messages between internal components and also between UAVs. While this architecture facilitated research, it was not very flexible, and changes were difficult to make.

From 2006 to 2012, we implemented path planning and task assignment algorithms in a single-threaded C++ program on a ground station and used them to control real UAVs [8], [9]. Tasks were selected by a human ground station operator and sent to the software. The algorithms calculated waypoints that were sent to the UAVs. This worked well, but the software was not easy to maintain, and it was difficult to add new tasks.

In 2012 we moved our algorithms onboard the UAVs and added UGSs [10]. This necessitated adding new autonomous capabilities and the ability to pass messages between UAVs and UGSs, which while successful, proved to be very difficult. In 2013 after utilizing the legacy software in a large field experiment, we decided that we needed to re-build the algorithms in a new software architecture, which became UxAS.

Lessons Learned: From this past work, we learned that it is difficult to add new capabilities in a software framework structured as a single monolithic program. In terms of incorporating new capabilities, adding new types of tasks was complicated because it required editing a variety of functions related to task handling and registration that were scattered throughout the code. Now, UxAS has a well defined API for task services and a specialized service to manage them. In terms of messaging, while our legacy software did use messages to implement task requests and send out waypoint-based plans, all messages were routed in a single loop and handled in a single main class, making it extremely large and unwieldy and limiting it to execution in a single thread. Similarly, communication with external entities such as UGSs and other UAVs was based on message passing, but without the well defined pub/sub communication pattern now employed by UxAS, message flow was difficult to understand and maintain. In terms of configuration, a central utility in the legacy software loaded an XML file and populated a large global data structure that was used by all capabilities, and its logic had to be modified every time a new capability was added. The restructuring of capabilities into individual services allowed us to structure the XML file such that each XML element contains only information specific to a service, which is simply passed to the relevant service on initialization by a service manager.

B. Architecture Design

Based on our experience developing software for UAV autonomy, we wanted to design a new software framework that is flexible, understandable, reliable, and maintainable. To meet these objectives, we decided to construct a new software framework based on the following guiding principles:

minimal:	only include framework elements and capabilities that meet current requirements.
modular:	design all services to be self-contained.
distributed:	no central mission controller/coordinator.

Four aspects of the UxAS architecture are key for meeting these guiding principles: Communications, Messaging, Configuration, and Service Management.

1) Communications: Based on our guiding principles, we investigated a communication software ecosystem based on ZeroMQ or ØMQ [11]. ZeroMQ provides a mechanism for implementing multithreaded applications in a modular way across a variety of programming languages and operating systems. To this end, we designed UxAS to consist of modular "services" that run in their own threads and use ZeroMQ to send and receive messages. This is the only connection between services. UxAS uses a pub/sub pattern to distribute messages. Each service subscribes only to the messages necessary to perform its function and sends messages to the publisher, which in turn sends them to subscribed services. Note that while the pub/sub infrastructure acts as a local process communication bus, ZeroMQ network adapter services facilitate connecting communication buses between processes. For discovering, connecting, and communicating with other entities running UxAS, there is a discovery service based on Zyre [12], an open-source framework for proximity-based peer-to-peer applications. This service sets up a pub/sub bridge between connected entities.

2) Messaging: For messages, UxAS uses the Lightweight Message Construction Protocol (LMCP) [13]. LMCP was developed at AFRL to provide a mechanism to define message types as structured data, automatically generate message classes in a variety of programming languages, and automatically generate the serialization and deserialization routines necessary to send message objects between applications. Message types are defined in a Message Data Model (MDM), an XML-formatted file that defines the data fields each message type has. Data fields can be primitives such as booleans, bytes, integers, reals, and strings; other LMCP message types; or arrays of these. MDMs allow developers to create custom message sets for different applications. An example is the Common Mission Automation Services Interface (CMASI), which defines data relevant for mission-level UAV autonomy. CMASI is one of the main MDMs used by UxAS.

3) Configuration: In order to reduce programming overhead and keep services independent, each service defines and handles its own configuration element. UxAS itself is configured based on an element in an XML-formatted configuration file. Services are initialized and configured based on additional elements in the same file. Before a service is started, it is presented with its configuration element. The service can then parse the element and perform the necessary configuration actions. This helps to encapsulate programming of services, i.e. each service defines and implements its own configuration.

4) Service Management: Services are created, started, and stopped by a service manager. The service manager parses the configuration file and then, for each service entry, creates a new instance of the service, passes the configuration entry to that service, and starts the service. In order to stop the service, the service manager sends a *KillService* message. The service receives this message, exits its main loop and performs any clean-up required before calling its class destructor.

IV. How does UxAS compare to other Software Frameworks

The structure of UxAS is similar to many Robotics Software Frameworks (RSFs) [14] in that most implement autonomous software components that perform duties within the framework by sending and receiving messages. Note that UxAS calls these software components *services* while many other frameworks call them *nodes*. In order to compare UxAS with other software frameworks, the next section defines common RSF elements. After that is a direct comparison of UxAS and a popular RSF, the Robot Operating System (ROS) [15].

A. RSF Elements

In [14], the authors relate RSFs to multi-agent systems, describe the important elements of RSFs, and compare several RSFs. What follows is a brief summary of UxAS in terms of some of those key elements (where "*nodes*" generally equate to UxAS's "*services*"):

Node Communication Mechanisms:

- ***Simple Messages*:** one-to-one messaging. One node sends an asynchronous message and the other receives it. *UxAS implements Simple Messages through the use of "Node ID"-based addresses.*
- ***Ports*:** one-to-many messaging. Nodes write messages to *out-Ports* that are connected to one or many *in-Ports* on other nodes. *UxAS does not implement Ports.*
- ***Topics*:** asynchronous many-to-many messaging implemented using a pub/sub pattern. Nodes publish messages to topics, and all nodes that have subscribed to a topic receive the messages published to it. *UxAS implements Topics based on LMCP message types.*
- ***Events*:** one-to-many messaging. Similar to ports, but without a connection. *UxAS does not use Events.*
- ***Services*:** allow remote execution of a procedure using a request/reply pattern. *UxAS implements this functionality through the use of addressing based on Receiver ID and Sender ID. A message is sent to the Receiver ID and, once the receiver has executed the procedure, it sends the results back to using the Sender ID.*
- ***Properties*:** a method to set and get node parameters. *UxAS does not implement Properties; services are configured only at initialization.*

Naming Service: global service that allows the localization of nodes and other global resources such as topics from a name. *UxAS does not implement a Naming Service.*

Lookup Service: global service that provides a directory of existing resources in the system. *UxAS does not implement a Lookup Service.*

Discovery Service: distributed service that enables a node that offers a certain service to be found. *UxAS implements a Discovery Service that finds and connects to other entities running the service.*

Multi-Agent or Robotics Standard: standards that define agent interaction patterns, their communication language, and the necessary system infrastructure. *UxAS's only standard is the LMCP message format.*

Message Format and Marshaling: binary or message formats such as XML, XDR, CSV, JSON, YAML and methods for putting messages on the communication 'wire.' *UxAS defines Message Data Models in XML, and LMCP tools are used to generate message classes and routines to serialize and deserialize message objects.*

Concurrency Model: autonomous nodes either run in their own threads, own processes, or a combination of both. *UxAS services can run in their own threads or as separate processes. When they run as separate processes, they connect using one of a number of network bridge services.*

Message transport: communication protocols such as HTTP, RMI, CORBA, TCP, and UDP. *UxAS makes use of the ZeroMQ ecosystem of software to implement connections through shared memory, IPC, UDP, and TCP. UxAS also implements communication over serial connections.*

Hardware Interfaces and Drivers: the method of supporting hardware and other interfaces. While this varies, most RSFs use one of the following approaches:

- ***API isolation*:** utilizes a programming interface such as a dynamic link library or an RPC-API server.
- ***Node isolation*:** drivers are integrated into a node. This makes it possible to communicate to the interface from other nodes. *This is the method UxAS uses to interface with autopilots, sensors, and other networks.*

Real-Time Capabilities: the ability to support hard real-time constraints. *UxAS does not have Real-Time Capabilities. While there are timing requirements for the mission management and teaming functions implemented in UxAS, they do not require hard real-time synchronization.*

Introspection and Management Tools: permit the monitoring, visualization, and analysis of the state of the system. *Since UxAS makes extensive use of messages to perform operations, UxAS provides a service that captures and logs desired messages in a database for evaluation.*

Robotics Algorithms: provision of generic and reusable robotic algorithms. *UxAS provides baseline algorithms for multi-vehicle mission autonomy.*

Simulation and Modeling: tools used to develop, test, and debug systems and also exercise candidate systems in a controlled environment. *UxAS uses software tests to examine individual services. The AMASE simulation[2] can be used to validate UxAS with simulated UAVs.*

B. UxAS Comparison to ROS

ROS is an RSF that has gained a wide following in the robotics community [16]. The focus of ROS is "a framework for writing robot software" [17], whereas the focus of UxAS is mission-level autonomy for single UAVs and UAV teams. So while both are likely capable of providing the same capabilities, it should generally be easier to implement multi-UAV autonomy in UxAS. In order to better understand the difference between UxAS and ROS, it is useful to note the different design considerations in terms of implementation, navigation, motion control, processing capabilities, sensing, and communications.

implementation: ROS was designed to facilitate building flexible robotic systems with reusable software components and distributed processing. UxAS was designed to facilitate building flexible systems with reusable software components and UAV-centric processing that can be distributed over many UAVs.

navigation: In general, robots must sense their environment in order to make plans to move through it. UAV operating environments are generally well known, with pre-defined "keep-in" and "keep-out" zones.

motion control: Much of the complication of robot control is coordinating the systems that allow the robot to move through its environment. This is normally accomplished

[2]https://github.com/afrl-rq/OpenAMASE

using a distributed set of motion controllers. UAVs must maintain stability and control of the vehicle using aerodynamic surfaces, which is normally accomplished using a single autopilot controller.

processing capabilities: While the design of both robots and UAVs must accommodate SWaP requirements, UAVs are generally much more limited on the number and size of processors available.

sensing: robots generally use a stable platform to sense objects close to them. UAVs sense objects from a constantly changing frame-of-reference from relatively far away.

communications: Most research into robot teams assumes the robots are generally able to communicate. UAV missions in UxAS tend to cover large areas, greatly increasing the chance that UAVs will be out of communication range.

At a high level, the architectures of ROS and UxAS are very similar. Both are composed of independent software components – *nodes* for ROS and *services* for UxAS – that exchange information through message passing. The first key difference is that ROS *nodes* run in separate processes and UxAS *services* generally run in their own threads in a single process. Both of these architectures facilitate distributed components that are flexible and easy to maintain, but both result in different issues.

While both use peer-to-peer communication to exchange messages, ROS *nodes* negotiate individual network connections with other *nodes*. ROS makes use of a *master* node that implements naming, lookup, and parameter services. These services facilitate finding *nodes* that publish desired messages; using names to get the addresses of other *nodes* and streams; and managing configuration parameters. UxAS *services* on a single process running on one UAV communicate with each other by pushing messages to a common message relay that publishes those messages to any *service* that has subscribed to them. The message relays utilize ZeroMQ [11] sockets, which implement communication through shared memory. UAV-to-UAV communications either take place on prearranged TCP-IP network connections or over network connections that have been established through a UxAS *discovery service*. Since all UxAS *services* on a single UAV are connected to the message publisher, specialized naming, lookup, and parameters services are not required. When there are multiple UAVs in the system, the messages to be exchanged are predetermined.

Since ROS distributes its *nodes* among processes, special programs such as *rosrun*, *roslaunch*, *rostest*, *rosnode*, *rostopic*, and *rosmsg* were written to coordinate running the nodes and analyzing the behavior of the system. By implementing nodes in separate processes, it is straightforward to implement a ROS-based system on either one processor or distributed over many processors. The majority of UxAS *services* run in a single process; therefore, all of these services are started by core UxAS functions, and system analysis is facilitated by special message logging services.

Both ROS and UxAS messages are defined in message definition files that are compiled into language specific implementations. ROS message definition files are fairly simple, with lines consisting of field types and names. There is a separate file for each message. UxAS uses LMCP messages (see Section III-B2), which are organized into Message Data Models (MDMs) [13]. All message types in an MDM are specified in a single XML file. LMCP supports inheritance in message types, while ROS does not.

Since ROS distributes its *nodes* in separate processes, *nodes* can be written in any language supported by ROS messaging. With UxAS's UAV centric organization, *services* are written in C++. External services, written in any language supported by LMCP and ZeroMQ, can be connected through networking.

V. Configuring UxAS and Its Services

The behavior of UxAS depends on its own configuration parameters, as well as what services that are enabled and how they are configured. Services are configured using a global configuration file written in XML. The default configuration file is ***cfg.xml***, but a different file can be specified as a command line argument to UxAS, e.g.

uxas -cfgPath *../PathToConfigurationFile/cfgFileName.xml*

The elements contained in a UxAS configuration file include the **UxAS Element**, **Service Elements**, and **Bridge Elements**. The **UxAS Element** accepts the following attributes:

***EntityID*:** ID number of the entity represented by this instance of UxAS.

***EntityType*:** used to differentiate between entities such as *Aircraft* and *UGS*. Attribute types and values are defined by the services that use them.

***ConsoleLoggerSeverityLevel*:** if present, all log messages at or below the severity level are displayed in the console. Valid entries are: *DEBUG*, *INFO*, *WARN*, and *ERROR*.

***MainFileLoggerSeverityLevel*:** if present, all log messages at or below the severity level are saved in log files. Valid entries are: *DEBUG*, *INFO*, *WARN*, and *ERROR*.

***RunDuration_s*:** UxAS will run for *RunDuration_s* seconds before terminating.

Service Elements determine what services are loaded and how they are configured. There can be as many Service Elements as required. The attributes of a Service Element are the options defined by the corresponding service. For example, the HelloWorld service can be configured with the following Service Element:

```xml
<Service Type="HelloWorld" StringToSend="Hello␣from␣#1"
    SendPeriod_ms="1000"/>
```

Bridge Elements configure bridges, which are services that create communication connections. For example, a Zyre connection can be configured with the following Bridge Element:

```xml
<Bridge Type="LmcpObjectNetworkZeroMqZyreBridge"
    NetworkDevice="enx281878b9065a"> ... </Bridge>
```

VI. Core Services

UxAS is configurable and extensible, meaning it can run in a variety of ways. However, UxAS provides a core set of services for multi-UAV mission-level autonomy. These services work in concert to carry out what we call the *task assignment pipeline*, which makes some assumptions about how services interact with each other. This pipeline is best understood in the context of the basic ISR mission of Section II-A and is depicted as a sequence diagram in Figure 3. We start by providing an overview of this pipeline, then provide a detailed explanation of each of the involved services. Documentation of LMCP messages in this section can be found in the UxAS repository.

A. *Overview of the Task Assignment Pipeline*

UxAS uses AutomationRequest messages to kick off the task assignment pipeline. An AutomationRequest includes the IDs of tasks to be performed, the IDs of vehicles eligible to perform them, the ID of an operating region (consisting of keep-in and keep-out zones), and optional constraints on the order in which tasks can be assigned. In order for an AutomationRequest to be valid, all entities referenced in the request must have been previously defined though the appropriate messages, e.g. AirVehicleConfiguration, AirVehicleState, KeepInZone, KeepOutZone, OperatingRegion, and Task messages. These messages can be sent in from an external application, or there is a *SendMessagesService* that can be used to send in messages stored as external files at specified times. The *AutomationRequestValidatorService* checks whether all entities referenced in an AutomationRequest have been defined. If so, it attaches a unique identifier to the request and starts the task assignment pipeline. If additional AutomationRequest messages are received, they are queued until the pipeline is finished.

Tasks referenced in an AutomationRequest must be defined before the request is made. For each type of task, there is an LMCP message that inherits from the base LMCP message Task, e.g. LineSearchTask, and there is a service class that implements the logic needed to plan and carry out the task, e.g. *CmasiLineSearchTaskService*. When a Task message is received, the *TaskManagerService* instantiates an object of the corresponding task class and passes in configuration information contained in the Task message, e.g. the points comprising a line for a particular LineSearchTask and the TaskID. The task then sends a TaskInitialized message. We emphasize that there can be multiple instantiations of the same *Task* service class, e.g. there will be two different *CmasiLineSearchTaskService*s running if there are two different LineSearchTasks.

Tasks generally have options on how they can be performed, e.g. a LineSearchTask has options on which end of the line defines the start of the task and what yaw angles are allowed for the sensor. When a *Task* sees its TaskID referenced in a UniqueAutomationRequest, it computes task-level waypoints for each of the available options and eligible vehicles, possibly using information from *SensorManagerService*s for sensors aboard the vehicles to reason about the size and positioning of sensor footprints and available sensor types. At this stage, task-level waypoints and options might be approximate, e.g. because options for the approach angle for an unconstrained PointSearchTask were generated by discretizing the full range of possible angles by increments of 20 degrees.

Once a task has computed a set of task-level waypoints for an option, it publishes a series of RouteRequest messages that are received by the *RouteAggregatorService*, which uses the *RoutePlannerVisibilityService* to compute on-task routes and costs for reaching the task-level waypoints while satisfying operating region constraints. Note that if task-level waypoints are only approximate, the task can generate more refined routes later, so the current RouteRequest might be for costs only (denoted with a * in Figure 3). The *RouteAggregatorService* then returns on-task routes and/or costs to the *Task*. The *Task* then narrows down the options by removing options with routes that would violate the operating region, vehicles that do not have appropriate sensor types, excessive cost, etc. It then publishes a TaskPlanOptions message with the final set of options. The *RouteAggregatorService* then queries the *RoutePlannerVisibilityService* to determine enroute vehicle-to-task costs (i.e. for each vehicle to reach the starting point of each task) and enroute task-to-task costs (i.e. to go from the end of each task to the start of every other task).

Costs are passed to the *AssignmentTreeBranchBoundService*, which determines the vehicle assignment and task ordering that minimizes cost while adhering to an optional process algebra string [18] specified in the AutomationRequest. Process algebra has three operators. The *alternative* operator $a + b$ means assign task a or b, but not both. The *sequential* operator $a \cdot b$ means assign task a followed by task b. The *parallel* operator $a \parallel b$ means assign task a and b in any order. Longer process algebra strings are built inductively from these operators; in fact, for each task, all final task plan options are aggregated into a process algebra substring using the alternative operator. Note that in UxAS, process algebra constraints currently apply to the order in which tasks are assigned, but not necessarily the order in which they execute. For instance, if UAVs uav_1 and uav_2 are eligible to perform tasks a and b with the constraint $a \cdot b$, a could be assigned to uav_1, then b could be assigned to uav_2. However, b might be executed before a, e.g. if uav_2 is closer to b than uav_1 is to a at the start of the mission. This service then publishes a TaskAssignmentSummary that encodes the assignment.

The *PlanBuilderService* receives the TaskAssignmentSummary and sends out a TaskImplementationRequest for each task individually in the order dictated by the assignment. When a *Task* sees its TaskID in a TaskImplementationRequest, it must respond with a TaskImplementationResponse that includes its enroute and on-task waypoints. Since earlier route planning might have been based on approximate options, a task can now refine its route, making as many rounds of RouteRequests as necessary to create the final enroute and on-task waypoints for the TaskImplementationResponse. The *PlanBuilderService* collects the TaskImplementationResponses, and once it has them all, it creates a MissionCommand for each vehicle that covers all the vehicle's assigned tasks, and the MissionCommands are aggregated into an AutomationResponse. Each MissionCommand contains the list of waypoints the assigned vehicle should fly. Each UAV running UxAS publishes periodic AirVehicleState messages that report the Waypoint the UAV is headed to and the AssociatedTasks that generated it. Once the UAV reaches a waypoint, AirVehicleStates start reporting the next Waypoint and its AssociatedTasks. Once a *Task* sees its TaskID listed in the AssociatedTasks of an AirVehicleState, it takes control of the vehicle, e.g. steering the sensor using VehicleActionCommands.

B. Service Details

1) AutomationRequestValidatorService: This service provides a simple sanity check on external automation requests. It also queues requests and tags them with unique identifiers, feeding them into the system one at a time.

This service has two states: **idle** and **busy**. Certain non-AutomationRequest messages are relevant to this service, e.g. task, vehicle configuration, vehicle state, zone, and operating region messages. When such a message is received, its information is added to a local memory store.

Upon reception of an AutomationRequest message, this service ensures that the request can be carried out by checking its local memory store for existence of the requested vehicles, tasks, and operating region. If the request includes vehicles, tasks, or an operating region that has not previously been defined, this service will publish an error message.

Upon determination that the AutomationRequest includes only vehicles, tasks, and an operating region that have previously been defined, this service creates a UniqueAutomationRequest with a previously unused unique identifier. If in the **idle** state, this service will immediately publish the UniqueAutomationRequest message and transition to the **busy** state. If already in the **busy** state, the UniqueAutomationRequest will be added to the end of a queue.

When this service receives either an error message (indicating that the UniqueAutomationRequest cannot be fulfilled) or a corresponding UniqueAutomationResponse, it will publish the same message. If in the **idle** state, it will remain in the **idle** state. If in the **busy** state, it will remove the fulfilled request from the queue and then send the next UniqueAutomationRequest in the queue. If the queue is empty, this service transitions back to the **idle** state.

This service also includes a parameter that allows an optional *timeout* value to be set. When a UniqueAutomationRequest is published, a timer begins. If the *timeout* has been reached before a UniqueAutomationResponse is received, an error is assumed to have occurred, and this service removes the pending UniqueAutomationRequest from the queue and attempts to send the next in the queue or transition to **idle** if the queue is empty.

2) Task: A *Task* forms the core functionality of vehicle behavior. It is the point at which a vehicle (or set of vehicles) is dedicated to a singular goal. Each *Task* service is responsible for estimating its cost during the early planning phase of the task assignment pipeline, then controlling the vehicle during execution of the particular task. During *Task* execution, a wide spectrum of behavior is allowed. This includes updating waypoints and steering sensors, potentially deviating from what was used to estimate cost during the planning phase. As part of the core services, this general *Task* description stands in for all *Task* service classes that can run in the system.

The general *Task* interaction with the rest of the task assignment pipeline is complex. It is the aggregation of each *Task* service's possibilities that defines the complexity of the overall mission assignment. These *Task* possibilities are called *options*, and they describe the precise ways that a *Task* could unfold. For example, a LineSearchTask could present two options to the system: 1) search the line from East-to-West and 2) search the line from West-to-East. Either is valid, and selecting the option that optimizes overall mission efficiency is the role of the assignment service.

A general *Task* comprises up to nine states with each state corresponding to a place in the message sequence that carries out the task assignment pipeline. The states for a *Task* are:

- **Init**: This is the state that all *Tasks* start in until all internal initialization is complete. For example, a *Task* may need to load complex terrain or weather data upon creation and will require some (possibly significant) start-up time. When a *Task* has completed its internal initialization, it must report transition from this state via a TaskInitialized message.
- **Idle**: This represents the state of a *Task* after initialization, but before any requests have been made that include the *Task*. UniqueAutomationRequest messages trigger a transition from this state into the **SensorRequest** state.
- **SensorRequest**: When a *Task* is notified of its inclusion in a UniqueAutomationRequest (by noting the presence of its TaskID in the TaskList of the message), it can request calculations that pertain to sensors onboard the vehicles included in the UniqueAutomationRequest message. While waiting for a response from the *SensorManagerService*, a *Task* is in the **SensorRequest** state until the response from the *SensorManagerService* is received.
- **OptionRoutes**: After the *SensorManagerService* has replied with the appropriate sensor calculations, the *Task* can request waypoints from the *RouteAggregatorService* that carry out the on-*Task* goals. For example, an *AreaSearchTask* can request routes from key surveillance positions that ensure sensor coverage of the entire area. The *Task* remains in the **OptionRoutes** state until the *RouteAggregatorService* replies.
- **OptionsPublished**: When routes are returned to the *Task*, it will utilize all route and sensor information to identify and publish the applicable TaskPlanOptions. The determination of TaskPlanOptions is key to overall mission performance and vehicle behavior. It is from this list of options that the assignment will select in order to perform this particular *Task*. After publication of the options, a *Task* waits in the **OptionsPublished** state until the TaskImplementationRequest message is received, whereupon it switches to **FinalRoutes**.
- **FinalRoutes**: Upon reception of a TaskImplementationRequest, a *Task* is informed of the option that was selected by the assignment service. At this point, a *Task* must create the final set of waypoints that include both *enroute* and *on-task* waypoints from the specified vehicle

location. The *Task* is required to create *enroute* waypoints since a route refinement is possible, taking advantage of the concrete position of the selected vehicle. *On-task* waypoints can also be refined. The *Task* remains in the **FinalRoutes** state until the route request is fulfilled by the *RouteAggregatorService*.

- **OptionSelected**: When the final waypoints are returned from the *RouteAggregatorService*, the *Task* publishes a complete TaskImplementationResponse message. A *Task* will remain in this state until an AirVehicleState message includes this TaskID in its AssociatedTaskList. If during this state, a subsequent UniqueAutomationRequest is made, the *Task* returns to the **SensorRequest** state and immediately attempts to fulfill the requirements of the new UniqueAutomationRequest. This behavior implies that a *Task* can only be part of a single AutomationRequest, and subsequent requests always override previous requests.
- **Active**: If the *Task* is in the **OptionSelected** state and an AirVehicleState message is received that includes the TaskID in the list of AssociatedTasks, then the *Task* switches to the **Active** state and is allowed to publish new waypoints and sensor commands at will. A *Task* remains in the **Active** state until a subsequent AirVehicleState message does *not* list the TaskID in its AssociatedTasks. At which point, a transition to **Completed** is made. Note that a *Task* can relinquish control indirectly by sending the vehicle to a waypoint not tagged with its own ID. Likewise, it can maintain control indefinitely by ensuring that the vehicle only ever goes to a waypoint that includes its ID. If a UniqueAutomationRequest message that includes this TaskID is received in the **Active** state, it transitions to the **Completed** state.
- **Completed**: In this state, the *Task* publishes a TaskComplete message and then immediately transitions to the **Idle** state.

3) RoutePlannerVisibilityService: This service performs approximate route planning using a visibility graph. One of the fundamental architectural decisions in UxAS is separation of route planning from task assignment. This service is an example of a route planning service for aircraft. Ground vehicle route planning (based on Open Street Maps data) can be found in the *OsmPlannerService*.

The design of the *RoutePlannerVisibilityService* message interface is intended to be as simple as possible: a route planning service considers routes only in fixed environments for known vehicles and handles requests for single vehicles. The logic necessary to plan for multiple (possibly heterogeneous) vehicles is handled in the *RouteAggregatorService*.

In two dimensional environments composed of polygons, the shortest distance between points lies on the visibility graph. The *RoutePlannerVisibilityService* creates such a graph and, upon request, adds desired start/end locations to quickly approximate a distance-optimal route through the environment. With the straight-line route created by the searching the visibility graph, a smoothing operation is applied to ensure that minimum turn rate constraints of vehicles are satisfied. Note that this smoothing operation can violate prescribed keep-out zones and is not guaranteed to smooth arbitrary straight-line routes (in particular, path segments shorter than the minimum turn radius can be problematic).

Due to the need to search over many possible orderings of *Task*s during an assignment calculation, the route planner must quickly compute routes. Even for small problems, hundreds of routes must be calculated before the assignment algorithm can start searching over the possible ordering. For this reason it is imperative that the route planner be responsive and efficient.

4) RouteAggregatorService: The *RouteAggregatorService* fills two primary roles: 1) it acts as a helper service to make route requests for large numbers of heterogenous vehicles; and 2) it constructs the vehicle-to-task and task-to-task route-cost table that is used by the assignment service to order the tasks as efficiently as possible. Each functional role acts independently and can be modeled as two different state machines.

The *Aggregator* role orchestrates large numbers of route requests (possibly to multiple route planners). This allows other services in the system (such as *Task* services) to make a single request for routes and receive a single reply with the complete set of routes for numerous vehicles.

For every aggregate route request (specified by a RouteRequest message), the *Aggregator* makes a series of RoutePlanRequests to the appropriate route planners (i.e. sending route plan requests for ground vehicles to the ground vehicle planner and route plan requests for aircraft to the aircraft planner). Each request is marked with a RequestID and a list of all RequestIDs that must have matching replies is created. The *Aggregator* then enters a **pending** state in which all received plan replies are stored and then checked off the list of expected replies. When all of the expected replies have been received, the *Aggregator* publishes the completed RouteResponse and returns to the **idle** state.

Note that every aggregate route request corresponds to a separate internal checklist of expected responses that will fulfill the original aggregate request. The *Aggregator* is designed to service each aggregate route request even if a previous one is being fulfilled. When the *Aggregator* receives any response from a route planner, it checks each of the many checklists to determine if all expected responses for a particular list have been met. In this way, the *Aggregator* is in a different **pending** state for each aggregate request made to it.

The *RouteAggregatorService* also creates the AssignmentCostMatrix, which is a key input to the assignment service. For simplicity, this role will be labeled as the *Collector* role. This role is triggered by the UniqueAutomationRequest message and begins the process of collecting a complete set of on-task and between-task enroute costs.

The *Collector* starts in the **Idle** state and upon reception of a UniqueAutomationRequest message, it creates a list of TaskIDs that are involved in the request and then moves to the **OptionsWait** state. In this state, the *Collector* stores all TaskPlanOptions and matches them to the TaskIDs that were requested in the UniqueAutomationRequest. When all expected

TaskPlanOptions messages are received, the *Collector* moves to the **RoutePending** state. In this state, the *Collector* makes a series of requests for routes between 1) initial conditions of all vehicles to all tasks and 2) route plans between the end of each *Task* and start of all other *Task*s. Similar to the *Aggregator*, the *Collector* creates a checklist of expected route plan responses and uses that checklist to determine when the complete set of routes has been returned from the route planners. The *Collector* remains in the **RoutePending** state until all route requests have been fulfilled, at which point it collates the responses into a complete AssignmentCostMatrix. The AssignmentCostMatrix message is published and the *Collector* returns to the **Idle** state.

Note that the *AutomationValidatorService* ensures that only a single UniqueAutomationRequest is handled by the system at a time. However, the design of the *Collector* does allow for multiple simultaneous requests as all checklists (for pending route and task option messages) are associated with the unique RequestID from each UniqueAutomationRequest.

5) AssignmentTreeBranchBoundService: The *AssignmentTreeBranchBoundService* is a service that does the primary computation to determine an efficient ordering and assignment of *Task*s to available vehicles. The assignment algorithm reasons only at the cost level; in other words, the assignment itself does not directly consider vehicle motion but rather it uses estimates of motion cost. The cost estimates are provided by the *Task*s (for on-task costs) and by the *RouteAggregatorService* for task-to-task enroute travel costs.

The *AssignmentTreeBranchBoundService* can be configured to optimize based on cumulative team cost (i.e. sum total of time required from each vehicle) or the maximum time of final task completion (i.e. only the final time of total mission completion is minimized). For either optimization type, this service will first find a feasible solution by executing a depth-first, greedy search. Although it is possible to request a mission for which **no** feasible solution exists, the vast majority of missions are underconstrained and have an exponential (relative to numbers of vehicles and tasks) number of solutions from which an efficient one must be discovered.

After the *AssignmentTreeBranchBoundService* obtains a greedy solution to the assignment problem, it will continue to search the space of possibilities via backtracking up the tree of possibilities and *branching* at decision points. The cost of the greedy solution acts as a *bound* beyond which no other solution is be considered. In other words, as more efficient solutions are discovered, any partial solution that exceeds the cost of the current best solution will immediately be abandoned (cut) to focus search effort in the part of the space that could possibly lead to better solutions. In this way, solution search progresses until all possibilities have been exhausted or a pre-determined tree size has been searched. By placing an upper limit on the size of the tree to search, worst-case bounds on computation time can be made to ensure desired responsiveness from the *AssignmentTreeBranchBoundService*.

General assignment problems do not normally allow for specification of *Task* relationships. However, the *AssignmentTreeBranchBoundService* relies on the ability to specify *Task* relationships via process algebra constraints. This enables creation of moderately complex missions from simple atomic *Task*s. Adherence to process algebra constraints also allows *Task*s to describe their *option* relationships. All final task options are combined into a single process algebra substring or term using the alternative operator, which then replaces the task term in the original mission-level process algebra string. Due to the heavy reliance on process algebra specifications, any assignment service that replaces *AssignmentTreeBranchBoundService* must also guarantee satisfaction of such specifications.

The behavior of the *AssignmentTreeBranchBoundService* is straight-forward. Upon reception of a UniqueAutomationRequest, this service enters the **wait** state and remains in this state until a complete set of TaskPlanOptions and an AssignmentCostMatrix have been received. In the **wait** state, a running list of the expected TaskPlanOptions is maintained and updated. Upon receiving the AssignmentCostMatrix (which should be received strictly after the TaskPlanOptions due to the behavior of the *RouteAggregatorService*), this service conducts the branch-and-bound search to determine the proper optimized ordering and assignment of *Task*s to vehicles. The results are packaged into the TaskAssignmentSummary and published, at which point this service returns to the **idle** state.

6) PlanBuilderService: The final step in the task assignment pipeline is converting the decisions made by the *AssignmentTreeBranchBoundService* into waypoint routes that can be sent to each of the vehicles. Using the ordering of *Task*s and the assigned vehicle(s) for each *Task*, the *PlanBuilderService* will query each *Task* in turn to construct enroute and on-task waypoints to complete the mission.

Similar to both the *RouteAggregatorService* and the *AssignmentTreeBranchBoundService*, the *PlanBuilderService* utilizes a received UniqueAutomationRequest to detect that a new mission request has been made to the system. The UniqueAutomationRequest is stored until a TaskAssignmentSummary that corresponds to the unique RequestID is received. At this point, the *PlanBuilderService* transitions from the **idle** state to the **busy** state.

Using the list of ordered *Task*s dictated by the TaskAssignmentSummary, the *PlanBuilderService* sends a TaskImplementationRequest to each *Task* in order and waits for a TaskImplementationResponse from each *Task* before moving to the next. This is necessary as the ending location of a previous *Task* becomes the starting location for a subsequent *Task*. Since each *Task* is allowed to refine its final waypoint plan at this stage, the exact ending location may be different than what was originally indicated during the TaskPlanOptions phase. By working through the *Task* list in assignment order, all uncertainty about timing and location is eliminated, and each *Task* is allowed to make a final determination on the waypoints to be used.

Once all *Task*s have responded with a TaskImplementationResponse, the *PlanBuilderService* links all waypoints for each vehicle into a complete MissionCommand. The total set of MissionCommand messages are collected into the UniqueAutomationResponse which is broadcast to the system and represents a complete solution to the original AutomationRequest. At this point, the *PlanBuilderService* returns to the **idle** state.

VII. Utilization of UxAS

Even though UxAS is an experimental system, it has been used extensively on numerous projects over a period of 4 years to conduct small UAV testing. Since 2014, UxAS has been tested during 21 distinct events consisting of 88 sorties comprising nearly 150 flight hours. Projects included human operator efficiency analysis for control of multiple autonomous vehicles, ground-sensor-cued follow-on surveillance behaviors, base defense perimeter patrol patterns, onboard image analysis, formation flight, and a wide variety of cooperative autonomous behavior testing, including the missions of Section II and others.

At one event, UxAS was deployed simultaneously on 2 aircraft and 10 unattended ground sensors. During the course of that experiment, each instantiation of UxAS worked with the others in a decentralized manner to coordinate the detection of targets, the relay of detections to airborne assets, and the autonomous collection of overhead surveillance. These test events indicate that UxAS is flexible enough to accommodate autonomous behaviors across a variety of situations.

VIII. Conclusion

In this paper, we have introduced UxAS, an open-source software framework for mission-level autonomy for teams of unmanned systems. UxAS supports dynamic inter-UAV communication, provides a baseline set of autonomous capabilities, allows for the incorporation of new autonomous capabilities, and its service oriented architecture helps manage software complexity. We have provided a comparison of UxAS and other RSFs, in particular ROS. We have described the core services of UxAS and how they work together to plan and execute multi-vehicle missions, and we have discussed actual utilization and flight testing of UxAS.

In the future, we plan to focus on verification and validation (V&V) of UxAS. As stated in [19], "It is possible to develop systems having high levels of autonomy, but it is the lack of V&V methods that prevents all but relatively low levels of autonomy from being certified for use." To this end, many of our current and future research directions focus on V&V of UxAS and draw inspiration from formal methods, i.e. mathematically-based methods and tools for design and verification of software and hardware. In particular, we are working to formalize general safety requirements for unmanned systems and requirements for specific types of missions; to formalize UxAS's architecture in an architecture design language and verify that it has certain desired properties, e.g. that every UniqueAutomationRequest results in a UniqueAutomationResponse; to formalize requirements for individual services and prove that their implementations satisfy the requirements; to reimplement portions of UxAS in memory-safe languages such as Rust and Ada; and to run UxAS on a verified microkernel such as seL4. We anticipate that many different methods will be needed to provide sufficient evidence that UxAS meets its safety and performance requirements. We are therefore also developing an assurance case for UxAS, i.e. a documented body of evidence that provides a convincing and valid argument that a specified set of claims regarding a system's properties are adequately justified for a given application in a given environment [20].

References

[1] D. Kingston, S. Rasmussen, and L. Humphrey, "Automated UAV tasks for search and surveillance," in *Proc. IEEE Conf. Control Applications*, 2016, pp. 1–8.

[2] S. J. Rasmussen and T. Shima, "Tree search for assigning cooperating UAVs to multiple tasks," *International Journal of Robust and Nonlinear Control*, vol. 18, no. 2, pp. 135–153, 2008.

[3] H.-L. Choi, L. Brunet, and J. P. How, "Consensus-based decentralized auctions for robust task allocation," *IEEE Trans. on Robotics*, vol. 25, no. 4, pp. 912–926, 2009.

[4] S. Rasmussen, K. Kalyanam, and D. Kingston, "Field experiment of a fully autonomous multiple UAV/UGS intruder detection and monitoring system," in *Int. Conf. Unmanned Aircraft Systems*. IEEE, 2016, pp. 1293–1302.

[5] P. R. Chandler, S. J. Rasmussen, and M. Pachter, "UAV cooperative path planning," in *Proc. AIAA Guidance, Navigation, and Control Conf.*, 2000.

[6] C. Schumacher, P. R. Chandler, M. Pachter, and L. Pachter, "UAV task assignment with timing constraints via mixed-integer linear programming," in *Proc. AIAA Unmanned Unlimited Conf.*, 2004.

[7] T. Shima and S. Rasmussen, Eds., *Unmanned Aerial Vehicles, Cooperative Decision and Control: Challenges and Practical Approaches*, 1st ed., ser. Advances in Design and Control. Philadelphia, PA: SIAM, December 2008.

[8] D. B. Kingston and C. J. Schumacher, "Time-dependent cooperative assignment," in *Proc. American Control Conf.*, 2005.

[9] S. Rasmussen and D. Kingston, "Assignment of heterogeneous tasks to a set of heterogeneous unmanned aerial vehicles," in *Proc. AIAA Guidance, Navigation, and Control Conf.*, 2008.

[10] ——, "Development and flight test of an area monitoring system using unmanned aerial vehicles and unattended ground sensors," in *Int. Conf. Unmanned Aircraft Systems*, 2015.

[11] P. Hintjens, *ZeroMQ*, 1st ed. Sebastopol, CA: O'Reilly Media Inc., March 2013.

[12] (2018, Jan) Zyre. [Online]. Available: https://github.com/zeromq/zyre

[13] M. Duquette, "The common mission automation services interface," in *Infotech@Aerospace, AIAA-2011-1542*, St. Louis, MO, March 2011.

[14] P. Iñigo-Blasco, F. Diaz-del Rio, M. C. Romero-Ternero, D. Cagigas-Muñiz, and S. Vicente-Diaz, "Robotics software frameworks for multi-agent robotic systems development," *Robotics and Autonomous Systems*, vol. 60, no. 6, pp. 803–821, 2012.

[15] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, 2009, p. 5.

[16] T. Foote. (2018, Jan) Celebrating 9 years of ROS, the robot operating system. IEEE Spectrum. [Online]. Available: https://spectrum.ieee.org/automaton/robotics/robotics-software/celebrating-9-years-of-ros

[17] M. Quigley, B. Gerkey, and W. D. Smart, *Programming Robots with ROS: a practical introduction to the Robot Operating System*. O'Reilly Media, Inc., 2015.

[18] S. Karaman, S. Rasmussen, D. Kingston, and E. Frazzoli, "Specification and planning of UAV missions: a process algebra approach," in *American Control Conference*. IEEE, 2009, pp. 1442–1447.

[19] W. Dahm, "Report on technology horizons: A vision for Air Force science & technology during 2010-2030," USAF, Tech. Rep., 2010.

[20] D. J. Rinehart, J. C. Knight, and J. Rowanhill, "Current practices in constructing and evaluating assurance cases with applications to aviation," 2015.

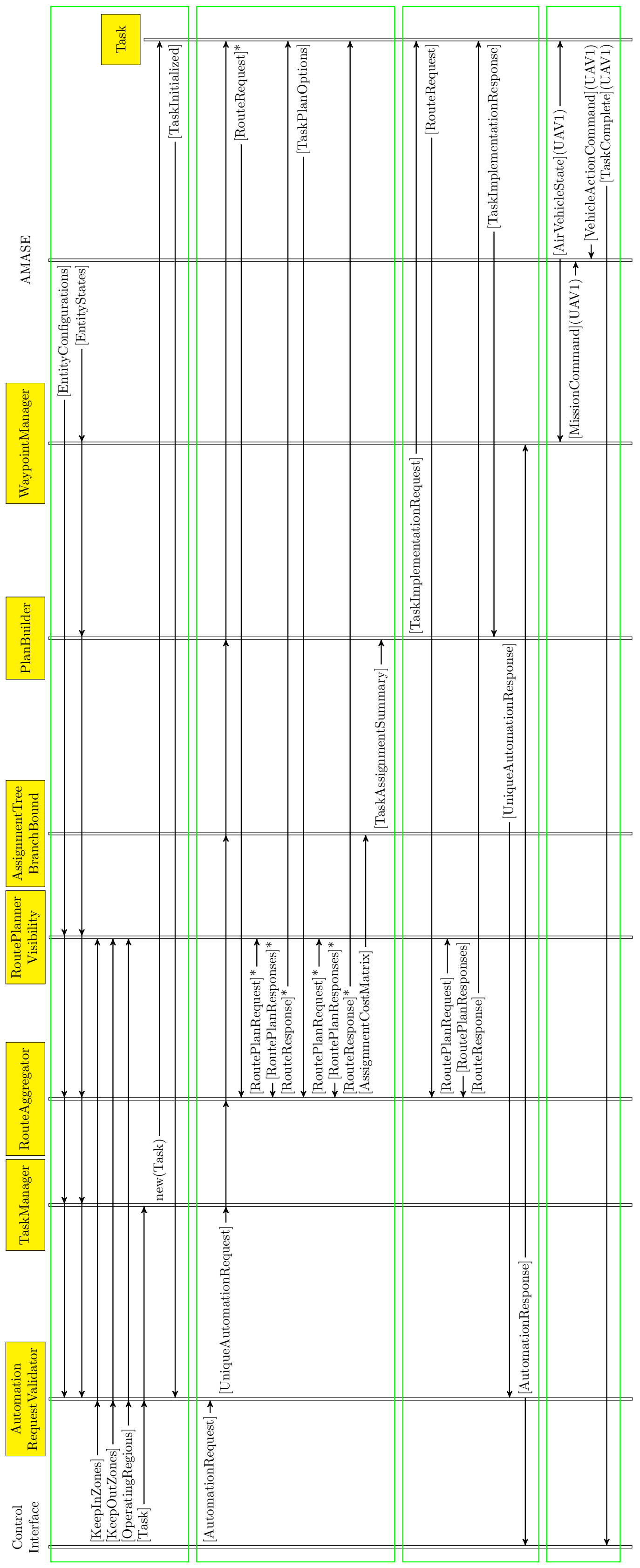

Fig. 3: A sequence diagram depicting major portions of UxAS's standard task assignment pipeline for basic ISR missions.