Mission Aware Flight Planning for Unmanned Aerial Systems

Eduard Santamaria, Pablo Royo, Cristina Barrado, Enric Pastor, and Juan López*
Technical University of Catalonia, Barcelona, Spain
Xavi Prats†
Technical University of Catalonia, Barcelona, Spain

The development of Flight Control Systems (FCS) coupled with the availability of other Commercial Off-The Shelf (COTS) components is enabling the introduction of Unmanned Aircraft Systems (UAS) into the civil market. UAS have great potential to be used in a wide variety of civil applications such as environmental applications, emergency situations, surveillance tasks and more. In general, they are specially well suited for the so-called D-cube operations (Dirty, Dull or Dangerous).

Current technology greatly facilitates the construction of UAS. Sophisticated flight control systems also make them accessible to end users with little aeronautical expertise. However, we believe that for its successful introduction into the civil market, progress needs to be made to deliver systems able to perform a wide variety of missions with minimal reconfiguration and with reduced operational costs.

Most current flight plan specification mechanisms consist in a simple list of waypoints, an approach that has important limitations. This paper proposes a new specification mechanism with semantically richer constructs that will enable the end user to specify more complex flight plans. The proposed formalism provides means for specifying iterative behavior, conditional branching and other constructs to dynamically adapt the flight path to mission circumstances. Collaborating with the FCS, a new module on-board the UAS will be in charge of executing these plans. The paper also presents a prototype implementation of this module and the results obtained in simulations.

Nomenclature

UAS	Unmanned Aircraft System
UAV	Unmanned Aerial Vehicle
USAL	UAS Service Abstraction Layer
RNAV	Area Navigation
FPMS	Flight Plan Manager Service
VAS	Virtual Autopilot Service
MCS	Mission Control Service
GCS	Ground Control Station
XML	Extensible Markup Language
KML	Keyhole Markup Language
FGFS	FlightGear Flight Simulator

*Computer Architecture Dept., Avda. del Canal Olímpic 15, 08860 Castelldefels, Spain.
†Castelldefels School of Technology, Avda. del Canal Olímpic 15, 08860 Castelldefels, Spain.

I. Introduction

Unmanned Aerial Systems (UAS) have been developed mainly in the military field. These systems have benefited from the development of Flight Control Systems (FCS), whose research started as a weapon feature introduced in many Air Force missiles, munitions and even in ejection seats. With the consolidation of FCS technologies and the incorporation of cheap Commercial Off-The-Shelf (COTS) electronics, the usage of UAS is now extending to the civil market. The availability of stable aircraft designs and autopilot systems greatly facilitates their construction. The sophistication of existing autopilots is also making these systems accessible to end users with little aeronautics expertise. However, much work remains to be done to deliver systems that can adapt to a wide array of missions with minimal reconfiguration and with reduced operational costs. A major effort has to be put in the formalism to define the aircraft behavior for pursuing the mission goals. While in most military missions the final objective is to reach a destination avoiding any obstacle[?] in most civil missions the final objective is surveillance of a given area.[?] The flight itself is only a means for gathering geographical information, more similar to a satellite mission, but with several advantages for end users' needs.

The introduction of UAS is clearly a new tool for remote sensing scientific applications. It has less costs compared to other aerial vehicles like conventional aircrafts or satellites. Among the benefits of using UAS there is the improvement of quality of the acquired geographical information. This is directly related to the quality (and cost) of sensors (i.e. CCD dynamic range, number of photodetectors, pixels resolution, in-camera imaging processing software, image file formats, etc.). It is not always possible to obtain the required image resolution from a low altitude satellite like the NASA TERRA satellite. With a cheaper sensor boarded on a UAS that flies three orders of magnitude lower, the resolution achieved can improve drastically.

But UAS have also the advantage of opportunity because end users can fly them whenever and wherever they need. Despite the huge number of orbiting satellites today, depending on their payload and position, the needed information for a given physical phenomena may not be available at the exact moment it is needed. Conventional aircrafts can be a solution to opportunity but only during light time, but not during darkness. Finally, safety and cost are the main reasons for the final introduction of UAS in the civil market. In dangerous situations and in highly repetitive surveillance operations the use of UAS is the most practical, flexible and economical solution.

This paper presents the advanced flying capabilities of a UAS as an extension of the FCS functionalities. Assuming a UAS with a FCS that ensures safe and stable maneuvers, we complement it with a user defined flight plan. The flight plan is characterized by offering semantically much richer constructs than those present in most current UAS autopilots,[1] which rely on simple lists of waypoints. This list of waypoints approach has several important limitations: it is difficult to specify complex trajectories and it does not support constructs such as conditional forks or iterations, small changes may imply having to deal with a considerable amount of waypoints and it provides no mechanism for adapting to mission time circumstances. To address these issues a new flight plan specification mechanism is proposed. The flight plan is organized as a set of stages, each one corresponding to a different flight phase. Each stage contains a structured collection of legs. The leg concept is inspired by current practices in Area Navigation (RNAV[2,3]). A leg describes the path the aircraft has to follow in order to reach a given destination waypoint. The leg concept is extended to accommodate higher level constructs for specifying iterations and forks. An additional leg type, referred to as parametric leg, is also introduced. The trajectory defined by a parametric leg is automatically generated as a function of mission variables, enabling dynamic behavior and providing a very valuable means for adapting the flight to the mission evolution. Another level of adaption is provided by the conditions governing the decision-making in intersection legs and the finalization of iterative legs.

In order to process and execute the user defined flight plan, a flight plan manager module is also proposed. The flight plan manager will interact with the on-board FCS to direct the aircraft according to the prescribed flight plan. A prototype of this module has been developed and tested as proof of the validity of the concept. In this paper we will focus on the development of fire fighting missions using UAS[4–6] as the motivating mission. We believe that the flight plan specification mechanism together with the flight plan manager will facilitate both the flight plan definition and execution processes. As a result the end user will be able to focus on the mission goals, i.e. the acquisition of useful data.

The organization of the paper is as follows: Section II presents the network centric architecture of the system and the middleware layer that will facilitate communication among payload components. In this section the UAS Service Abstraction Layer (USAL) is also described. As it will be seen, the USAL provides a standardized set of interfaces that isolate the system from the particular details of each piece

of equipment. Section III details the flight plan specification formalism. Section IV describes the flight plan manager capabilities and its operation. Section V presents the simulation environment and a mission scenario along with the simulation results. Finally, section VI concludes the article and identifies some future developments.

II. System Overview

This section describes the UAS architecture we propose[7] for executing civil missions: a distributed embedded system that will be on board the aircraft and that will operate as a payload/mission controller. Over the different distributed elements of the system we will deploy software components, called services, that will implement the required functionalities. These services cooperate to accomplish the UAS mission. They rely on a middleware[8] that manages and communicates the services. The communication primitives provided by the middleware promote a publish/subscribe model for sending and receiving data, announcing events and executing commands among services.

II.A. Distributed Embedded Architecture

The proposed system is built as a set of embedded microprocessors, connected by a Local Area Network (LAN), in a purely distributed and scalable architecture. This approach is a simple scheme which offers a number of benefits in our application domain that motivates its selection:

- Development simplicity is the main advantage of this architecture. Inspired by Internet applications and protocols, the computational requirements can be organized as services that are offered to all possible clients connected to the network.
- Extreme flexibility given by the high level of modularity of a LAN architecture. We are free to select the actual type of processor to be used in each LAN module. Different processors can be used according to functional requirements, and they can be scaled according to computational needs of the application. LAN modules with processing capabilities are referred to as nodes.
- Node interconnection is an additional extra benefit in contrast with the complex interconnection schemes needed by end-to-end parallel buses. While buses have to be carefully allocated to fit with the space and weight limitations in a small UAS, new nodes can be hot plugged to the LAN with little effort. The system can use wake-on-LAN capabilities to switch a node on when required during mission development.

II.B. Service Oriented Approach

Service Oriented Architectures (SOA) are getting common in several domains. For example, Web Services[9] in the Internet world, and Universal Plug and Play (UPnP)[10] in the home automation area. The main goal of a SOA is to achieve loose coupling among interacting components. We refer to the distributed components as services. A service is a unit of work, implemented and offered by a service provider, to achieve desired final results for a service consumer. Both provider and consumer are roles played by software agents on behalf of their owners.

The benefits of this architecture is the increment of interoperability, flexibility and extensibility of the designed system and of their individual services. In the implementation of a system, we want to be able to reuse existing services. SOA facilitates services reuse, while trying to minimize their dependencies by using loosely coupled services. Loose coupling among interacting services is achieved employing two architectural constraints. First, services shall define a small set of simple and ubiquitous interfaces, available to all other participant services, with generic semantics encoded in them. Second, each interface shall send, on request, descriptive messages explaining its operation and its capabilities. These messages define the structure and semantics provided by the services. The SOA constraints are inspired significantly by object oriented programming, which strongly suggests that you should bind data and its processing together.

In a network centric architecture like SOA, when some service needs some external functionality, it asks the system for the required service. If the system knows of a service that offers this capability, its reference is provided to the requester. Thus the former service can act as a client and consume that functionality using the common interface of the provider service. The result of a service interface invocation is usually the change

of state for the consumer but it can also result on the change of state of the provider or of both services. The interface of a SOA service must be simple and clear enough to be easy to implement in different platforms, both hardware and software. The development of services and specially their communication requires a running base software layer known as middleware.

II.C. Middleware

Middleware-based software systems consist of a network of cooperating components, in our case the services, which implement the logic of the application. The middleware is an integrating software layer that provides an execution environment and implements common functionalities and communication channels. On top of the middleware, services are executing. Any service can be a publisher, subscriber, or both simultaneously. This publish-subscribe model eliminates complex network programming for distributed applications. The middleware offers the localization of the other services and manages their discovery. The middleware also handles all the transfer chores: message addressing, data marshaling and demarshalling (needed for services executing on different hardware platforms), data delivery, flow control, message retransmissions, etc.

The main functionalities of the middleware are:

- Service management: The middleware is responsible of starting and stopping all the services. It also monitors their correct operation and notifies the rest of system about changes in service behavior.
- Resource Management: The middleware also centralizes the management of the shared resources of each computational node such as memory, processors, input/output devices, etc.
- Name management: The services are addressed by name, and the middleware discovers the real location in the network of the named service. This feature abstracts the service programmer from knowing where the service resides.
- Communication management: The services do not access the network directly. All their communication is carried by the middleware. The middleware abstracts the network access, allowing the services to be deployed in different nodes. Communication links from air to ground are also transparently managed by the middleware.

II.D. Communication Primitives

For the specific characteristics of a UAS mission, which may have a number of services interacting many-to-many, we propose four communication primitives based in the Data Distribution Services paradigm. It promotes a publish/subscribe model for sending and receiving data, events and commands among the services. Services that are producing valuable data publish that information while other services may subscribe to them. The middleware takes care of delivering the information to all subscribers. Many frameworks have been already developed using this paradigm, each one contributing with new primitives for such open communication scenario. In our proposal we implement only a minimalistic distributed communication system in order to keep it simple and soft real-time compliant. Next, we describe the proposed communication primitives, which have been named as Variables, Events, Remote Invocations and File Transmissions.

Variables are used for transmitting structured, and generally short, information from a service to one or more services of the distributed system. A Variable may be sent at regular intervals or each time a substantial change in its value occurs. The relative expiration of the Variable information allows to send it in a best-effort way. The system should be able to tolerate the loss of one or more of these data transmissions. The Variable communication primitive follows the publication subscription paradigm.

Events also follow the publication-subscription paradigm. The main difference in front of Variables is that Events guarantee the reception of the sent information to all the subscribed services. The utility of Events is to inform of punctual and important facts to all interested services. Some examples can be error alarms or warnings, indication of arrival at specific points of the mission, etc.

Remote Invocation is an intuitive way to model one-to-one interactions between services. Some examples can be the activation and deactivation of actuators, or calling a service for some form of calculation. Thus, in addition to Variables and Events, the services can expose a set of functions that other services can invoke or call remotely.

In some cases, there also exists the need to transfer continuous media. This continuous media includes generated photography images, configuration files or services program code to be uploaded to the middleware. The File Transmission primitive is used basically to transfer long file-structured information between nodes.

II.E. UAV Service Abstraction Layer

The UAV Service Abstraction Layer (USAL) is the set of available services running on top of the middleware to give support to most types of remote sensing UAV missions. USAL can be compared to an operating system. Computers have hardware devices used for input/output operations. Every device has its own particularities and the OS offers an abstraction layer to access such devices in a uniform way. Basically, it publishes an Application Program Interface (API) which provides end-users with efficient and secure access to all hardware elements. The USAL considers sensors and in general all payload as hardware devices of a computer. The USAL is a software abstraction layer that gives facilities to end-users programs to access the UAV payload. The USAL also provides many other useful features designed to simplify the complexity of developing the UAV's application.

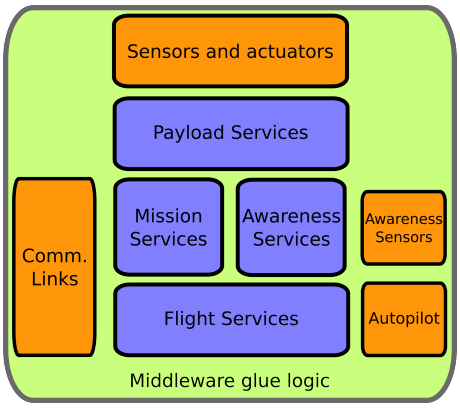


Figure 1. System Architecture.

The USAL defines a collection of reusable services that comprises a minimum common set of elements that are needed in most UAV missions. A number of specific services have been identified as "a must" in any real life application of UAVs (see Figure 1). The idea is to provide an abstraction layer that allows the mission developer to reuse these components and that provides guiding directives on how the services should interchange avionics information with each other. The available services cover an important part of the generic functionalities present in many missions. Therefore, to adapt our aircraft for a new mission it should be enough to reconfigure the services deployed to the UAV boards.

Available services have been classified in four categories:

1. Flight services: all services in charge of basic UAS flight operations: autopilot, basic monitoring, contingency management, etc.
2. Mission services: all services in charge of developing the actual UAS mission.
3. Payload services: specialized services interfacing with the input/output capabilities provided by the actual payload carried by the UAS.
4. Awareness services: all services in charge of the safe operation of the UAS with respect terrain avoidance and integration with shared airspace.

II.E.1. Flight Services

The flight services are a set of standardized architecture components designed to operate as an interface between the autopilot and the rest of subsystems in the UAS; e.g. the mission and payload services. The objective of the flight services is multiple:[11]

- Abstract autopilot details and peculiarities to the rest of the system.

- Extract internal sensor information from the autopilot and offer it to other services for its exploitation during the UAS mission.
- Provide a common flight-plan definition, improving by large actual commercial autopilot capabilities by adding a flight management layer on top of them.
- Provide status monitoring capabilities and automatic contingency management for efficient emergency response.

Figure 2 shows the fundamental components in the flight services category as well as the major relations among them.

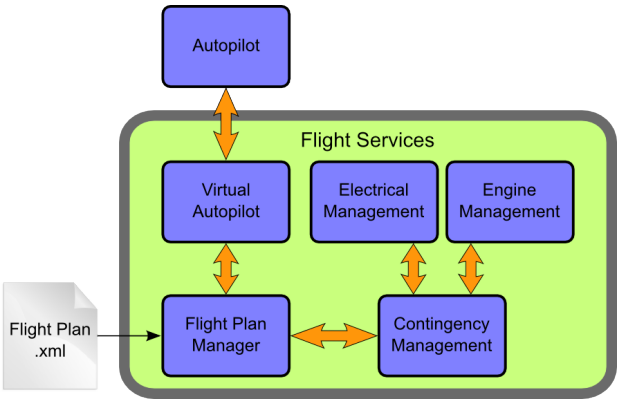


Figure 2. Overview of the available flight service category.

The Virtual Autopilot Service (VAS) is a service that on one side interacts with the selected autopilot and therefore needs to be adapted to its peculiarities. The VAS abstracts away the autopilot implementation details from its actual users. For other services in the UAS, the VAS is a service provider that offers a number of information flows to be exploited by them.

Given that not all autopilots are equal, the VAS follows a contract between the VAS as a service provider and its potential clients. This means that all the information provided by this service is standardized independently of the actual autopilot being used. However, the way this information is provided to its clients (event-based, continuous, and actual transmission rate) can be selected according to each individual client thanks to the communication infrastructure offered by the service-based architecture.

The flight planning capabilities of all autopilots are dissimilar, but generally limited to simple waypoint navigation. In some cases they include automatic take-off and landing modes. From the point of view of the actual missions or applications being developed by the UAS using a simple waypoint-based flight plan may be extremely restrictive. A Flight Plan Manager has been developed to implement much richer flight-plan capabilities on top of the available capabilities offered by the autopilot. The Flight Plan Manager Service offers structured flight plan definition using RNAV-inspired[3] legs as its main construction unit. Legs are organized in different stages with built-in emergency alternatives. Additional mission oriented legs with a high semantic level like repetitions, parameterized scans, etc are also provided. All available highly semantic legs can be modified by other services in the UAS by changing a number of configuration parameters without having to redesign the actual flight plan; thus truly allowing cooperation between the autopilot operation and the specific UAS mission.

Given that, in general, the real autopilot capabilities will be much simpler than those available in the flight plan manager, additional waypoints will be generated according to requirements. These internal waypoints will be dynamically fed into the autopilot through the VAS during mission time, therefore transforming the flight manager in a virtual machine capable of executing flight plans. As a result, combining both the abstraction mechanism provided by the VAS and the increased flight plan capabilities of the Flight Plan Manager Service, we obtain a highly capable platform that can be easily integrated to perform efficiently a number of valuable missions.

The Engine and Electrical Managers and the Contingency Manager are auxiliary services in charge of monitoring the engine and electrical parameters of the UAS and to collect status information related to multiple sources: engine, electrical, fuel, communications, etc; to detect if some type of contingency is occurring and decide which type of reaction is required.

III. Flight Plan Specification

Previous section has introduced the proposed UAS architecture. In this section we detail the specification mechanism that will be used to describe the UAS flight plans. Most current UAS autopilot systems rely on lists of waypoints as the mechanism for flight plan specification and execution. This approach has several important limitations: (1) It is difficult to specify complex trajectories and it does not support constructs such as forks or iterations. (2) It is not flexible because small changes may imply having to deal with a considerable amount of waypoints and (3) it is unable to adapt to mission circumstances. Besides (4) it lacks constructs for grouping and reusing flight plan fragments. In short, current autopilots specialize in low level flight control and navigation is limited to very basic go to waypoint commands. We believe that to improve current UAS operation higher level constructs, with richer semantics, and which enable flight progress to be aware of mission variables must be introduced. For that reason a new flight plan specification mechanism is proposed.

Some of the presented ideas are based on current practices in commercial aviation industry for the specification of RNAV[2] procedures. Area navigation (RNAV) is a method of navigation that takes advantage of the increasing amount of navigation aids (including satellite navigation) and permits aircraft operation on any desired flight path. RNAV procedures are composed of a series of smaller parts called legs. To translate RNAV procedures into a code suitable for navigation systems the industry has developed the Path and Termination concept. Path Terminator codes should be used to define each leg of an RNAV procedure. Leg types are identified by a two letter code that describes the path (e.g., heading, course, track, etc.) and the termination point (e.g., the path terminates at an altitude, distance, fix, etc.). Our specification mechanism makes use of the Path Terminator concept to describe basic legs. We are interested in a subset of RNAV legs applicable to GPS navigation. These elements are brought to the UAS field and extended with additional constructs. New control constructs such as iterative legs and intersection legs are added and adaptivity is increased by means of parametric legs.

This section identifies and describes the different elements that conform a flight plan specification for our UAS. The flight plan is specified in an XML document that will be submitted from the ground control station to the UAS in order to carry out its execution.

III.A. Flight plan document structure

The flight plan specification is stored in an XML document whose root node is *FlightPlan*. Listing 1 shows the elements contained within the root element. To make XML listings more readable some content has been replaced by ellipses.

Listing 1. XML flight plan document structure.

```
<FlightPlan xmlns='http://icarus.upc.es/schema/FlightPlan/1.0'>
   <Locale> ... </Locale>                    <!-- units and separators -->
   <Fixes> ... </Fixes>                      <!-- specific named locations -->
   <EmergencyPlans> ... </EmergencyPlans> <!-- emergency flight plans -->
   <MainFP> ... </MainFP>                    <!-- main flight plan -->
</FlightPlan>
```

Locale specifies what units are used for speed, altitude and distances. Also what are the decimal and group separators. *Fixes* contains a list of named waypoints, i.e. specific locations that, for some reason, are of special interest. *EmergencyPlans* contains a set of alternative plans in case an emergency occurs during execution of the main flight plan. And *MainFP* contains the main flight plan, that should be executed from beginning to end if no emergency occurs.

III.B. Fixes and Waypoints

A fix describes an specific location on the face of the earth. In commercial aviation fixes may refer to navigational aids, waypoints, intersections, airports, etc. In our case they refer to locations which, for some reason, are of special interest. As seen in listing 2 a fix has an identifier, a name and a description followed by its latitude and longitude coordinates.

Listing 2. List of fixes specification.

```
<Fixes>
   <Fix id="FIXID">
```

```
      <name>Example fix</name>
      <description>Some interesting place</description>
      <coordinates>37º38'0.0"N 122º22'0.0"W</coordinates>
   </Fix>
   <!-- More fixes may follow -->
</Fixes>
```

Waypoints are a key element of the flight plan description closely related to fixes. Conceptually a waypoint is a geographical position defined in terms of latitude/longitude coordinates. There are two kinds of waypoints, named waypoints and unnamed ones. The former correspond to fixes listed at the beginning of the flight plan, the latter are geographical positions with no association to any named location. Therefore, there are two ways of specifying waypoints, either by providing its coordinates or indicating the name of the fix it corresponds to. Apart from its location, a waypoint also has a type, which may be fly-by or fly-over. For fly-by waypoints passing close enough suffices while fly-over waypoints require passing upon them. Since changes of speed and altitude will also occur at specific waypoints, optionally a waypoint may also contain altitude and speed data. If these values are present, after getting to the waypoint the target speed and altitude of the aircraft will be set accordingly. Listing 3 shows an example of waypoint which refers to a fix. The *dest* element is part of a leg specification and is described in section III.E.

Listing 3. XML description of a waypoint.

```
<dest>
   <fix>FIXID</fix>
   <fly-over>true</fly-over>
   <altitude>300</altitude>
   <speed>65</speed>
</dest>
```

The rest of this section explains how a number of higher level constructs are put on top of the waypoint concept for providing a semantically richer flight plan specification. These constructs are presented in a top-down fashion starting with those located higher in the flight plan structure.

III.C. Emergency and Main Flight Plans

A flight plan specifies the path followed by the aircraft. Each flight plan is composed of a sequence of stages, such as take-off, departure procedure and others, which must come in correct order. Each flight plan stage is made up of a structured collection of legs. The leg concept is borrowed from RNAV and is used to specify the trajectory followed by the aircraft to reach a given waypoint from the preceding one. In the simplest case this trajectory will be a straight line.

All flights require a single main flight plan, but additional emergency plans may be present. Emergency flight plans are partial plans, i.e. they lack some initial stages, whose purpose is to provide alternative courses when an emergency situation occurs. Apart from the number of stages included, the main flight plan and the emergency plans have identical structure.

All flight plans have an identifier, a name and a description (see listing 4). Optionally, for the main flight plan a default emergency plan can be specified which will be superseded by emergency plans specified at stage or leg levels.

Listing 4. XML description of main flight plan.

```
<MainFP id="FPID">
   <name>Name of the flight plan</name>
   <description>Text describing the flight plan</description>
   <!-- List of stages that form the flight plan follows -->
   <stages> ... </stages>
   <emergency>EmergencyFPID</emergency>
</MainFP>
```

III.D. Stages

Stages organize legs and constitute high-level building blocks for flight plan specification. Each stage corresponds to a flight phase and groups together a collection of legs that seek a common purpose. Every stage, except for the first and last stages, has a single predecessor and a single successor. A stage may have more than one final leg. For instance, a take off procedure may end at different points depending on the selected

take off direction. Also, a stage may have more than one initial leg as could be the case for a departure procedure which could start at different positions depending on the take-off direction chosen. Usually there will be a one-to-one correspondence between the final legs of a given stage and the initial legs of the next one. Thus providing a seamless transition between stages. There are constructs (see section III.E.3) that enable the flight plan designer to provide this one-to-one correspondence if necessary. Because an emergency can occur at any point within a given stage, the transition to an emergency flight plan is an exception to the previous rule. Besides, the first stage of an emergency plan may have more than one initial leg. The selected leg to enter the emergency plan will be the one which is closest to the current aircraft position.

Listing 5. XML description of flight plan stage.

```
<stage id="STID" type="Departure" manualOnly="false">
   <name>Name of the stage</name>
   <description>Text describing the stage</description>
   <legs> ... </legs>                    <!-- Legs that belong to this stage  -->
   <initialLegs>LStart</initialLegs>     <!-- Space separated list of leg ids -->
   <finalLegs>LEnd</finalLegs>           <!-- Space separated list of leg ids -->
   <emergency>EmergencyFPID</emergency>  <!-- Emergency flight plan           -->
</stage>
```

As seen in listing 5, each stage has an identifier, a name and a description. A stage type attribute specifies its purpose, see table 1 for valid values. Another attribute indicates whether the stage must be executed manually. This is the case for taxi stages, whose automatic execution is not yet supported, but other parts of the flight plan (e.g. take-off and landing) could be manual-only as well. When such a stage type is encountered it is responsibility of an on-ground human pilot to control the aircraft. Additional elements indicate what are the sets of initial and final legs of the stage respectively. All stages may include an element indicating which emergency flight plan is to be carried out when an emergency occurs. This emergency plan will lead to a near area where landing is possible. If another emergency plan is specified at leg level, the latter will prevail.

Table 1. Stage types

Taxi	Move to or return from runway.
TakeOff	Legs in this stage will be used during a take off procedure.
Departure	These legs must be flown after taking off in order to reach the starting point of the next stage.
EnRoute	Navigate from an initial point to a destination point. It may appear more than once: from departure to mission site, from mission site to next mission site (if there is any) and from mission site to landing site.
Mission	Series of legs that will be flown during main mission operations.
Arrival	Legs to be flown after leaving the route and before initiating an approach procedure.
Approach	Prepare for landing.
Land	Landing operation.

III.E. Legs

A leg specifies the flight path to get to a given waypoint. In general, legs contain a destination waypoint and a reference to their next. Most times legs will be flown in a single direction, but within iterative legs (see section III.E.2) reverse traversal is also supported. In this case a reference to the previous leg will be present too. Only intersection legs, which mark decision points, are allowed to specify more than one next and previous legs.

There are four different kinds of legs:

- Basic legs: Specify leg primitives such as 'Direct to a Fix', 'Track to a Fix', etc.
- Iterative legs: Allow for specifying repetitive sequences.

- Intersection legs: Provide a junction point for legs which end at the same waypoint, or a forking point where a decision on what leg to fly next can be made.
- Parametric legs: Specify legs whose trajectory can be computed given the parameters of a generating algorithm, e.g. a scan pattern.

Intersection legs differ from the rest in that they may be reached from more than one predecessor and may lead to more than one successor. All legs have an optional parameter indicating what emergency flight plan is to be carried out when an emergency occurs.

III.E.1. Basic Legs

This section describes the basic legs available to the flight plan designer. They are referred to as basic legs to differentiate them from control structures like iterative or intersection legs and parametric legs. All of them are based on already existing ones in RNAV. Its original name is preserved.

Initial Fix (IFLeg) Determines an initial point. It is used in conjunction with another leg type (e.g. TF) to define a desired track.

Track to a Fix (TFLeg) Corresponds to a straight trajectory from waypoint to waypoint. Initial waypoint is the destination waypoint of the previous leg. Listing 6 shows how a Track to a Fix leg looks like in the XML flight plan description. The *xsi:type* attribute of the leg element identifies the leg type. *dest* is the destination waypoint. In the example a change of aircraft speed upon reaching this waypoint is requested.

Listing 6. XML description of Track to a Fix leg.

```
<leg id="L1" xsi:type="TFLeg">
   <dest>
      <coordinates>41°17'38.38"N 2°4'35.82"E</coordinates>
      <fly-over>true</fly-over>  <!-- Fly-over waypoint      -->
      <speed>60</speed>          <!-- Target speed after wp -->
   </dest>
   <next>L2</next>               <!-- Next leg id -->
</leg>
```

Direct to a Fix (DFLeg) Is a path described by an aircraft's track from an initial area direct to the next waypoint, i.e. fly directly to the destination waypoint whatever the current position is.

Radius to a Fix (RFLeg) Is defined as a constant radius circular path around a defined turn center that terminates at a waypoint. It is characterized by its turn center and turn direction.

Holding Pattern Specifies a holding pattern path. There are three kinds of holding patterns: Hold to an Altitude (HALeg), Hold to a Fix (HFLeg) and Hold to a Condition (HCLeg). In all cases the initial waypoint, the course (azimuth) of the holding pattern and the turn direction must be specified. The distance between both turn centers (d1) and the diameter of the turn segments (d2) are also needed. The three available types differ in how they are terminated. Hold to an Altitude terminates when a given altitude is reached, therefore the target altitude and the climb rate must be indicated. A Hold to a Fix is used to define a holding pattern path, which terminates at the first crossing of the hold waypoint after the holding entry procedure has been performed. The final possible type is the Hold to a Condition. In this case the holding pattern will be terminated after a given number of iterations or when a given condition no longer holds (regardless of the number of iterations).

III.E.2. Iterative Legs

A complex trajectory may involve iteration, thus the inclusion of iterative legs. An iterative leg has a single entry (i.e. its body can be entered from a single leg), a single exit and includes a list with the legs that form its body. Every time the final leg is executed an iteration counter will be incremented. When a given count is reached or an specified condition no longer holds the leg will be abandoned proceeding to the next one.

Figure 3 shows the two different possibilities for iterative leg specification. Diagram (A) displays the case when holds to a fix are used to reverse the aircraft course and cycle back and forth. After entering the iterative leg, the legs forming its body are executed. Then a hold to a fix is found which reverses the aircraft direction. Now the body legs can be executed again, but in reverse direction, until another hold to a fix is found. In the holding patterns, the solid line represents the path followed by the aircraft in order to perform the turning maneuver. This back and forth behavior is only allowed when it is possible to obtain the inverse of all legs involved. Diagram (B) shows a simpler case when the legs of the body are executed one after another in a single direction.

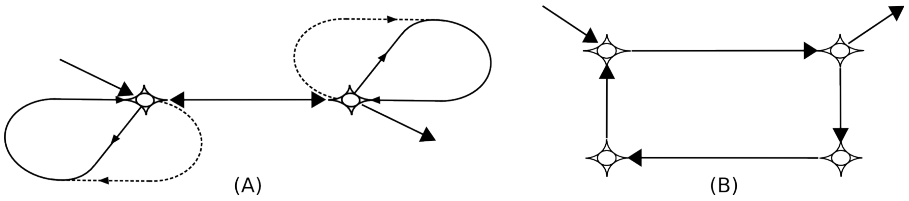


Figure 3. Iterative leg types.

Listing 7 shows the XML description of an iterative leg. There is a leg id and a reference to the next leg. Then the *body* element is found. This element contains the list of all legs that form the body of the iterative leg. *first* indicates which is the first leg to be executed upon entering the iterative structure and *last* which is the last leg executed before exiting. *upperBound* indicates how many times the iterative leg will be executed. It can also be exited before reaching this upper bound if its optional condition returns false.

Listing 7. XML description of an iterative leg.

```
<leg id="Loop" xsi:type="IterativeLeg">
   <next>L2</next>
   <body>LB1 LB2 LB3 LB4</body>  <!-- Body of the loop          -->
   <first>LB1</first>            <!-- First body leg           -->
   <last>LB3</last>              <!-- Last leg before exiting  -->
   <upperBound>5</upperBound>    <!-- Repeat five times        -->
   <cond>CondID</cond>           <!-- Condition that controls termination -->
</leg>
```

III.E.3. Intersection Legs

Intersection legs (see figure 4) indicate points where two or more different paths meet and where decisions on what to do next can be made. All joins and forks will end and start at an intersection leg.

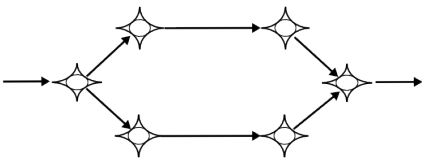


Figure 4. Intersection legs.

As seen in listing 8 an intersection leg contains a *next* element, a list of possible paths and a condition. The chosen path will be determined by the result of the condition and *next* is interpreted as a default value. If the condition is not set *next* is taken. If the intersection leg belongs to an iterative leg where backwards traversal is possible another list of alternatives along with its default path and condition must be present.

Listing 8. XML description of an intersection leg.

```
<leg id="Inter" xsi:type ="IntersectionLeg">
   <next>Alt1</next>              <!-- Selection     -->
   <nextList>Alt1 Alt2</nextList> <!-- Alternatives  -->
   <nextCond>CondId</nextCond>    <!-- Condition governing selection -->
</leg>
```

III.E.4. Parametric Legs

In many occasions the dynamic characteristics of the mission environment will make previous leg types insufficient. Parametric legs provide an increased level of adaption to changes that occur during mission time. With parametric legs the flight path is dynamically generated according to input values. Figure 5 shows a number of possible patterns for exploring a given area, as in (A) and (B), or a more specific point, as in (C).

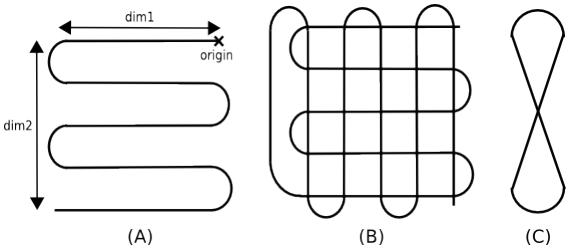


Figure 5. Scanning patterns.

Listing 9 shows a description of the parametric leg seen in figure 5-A. The *origin*, dimensions and *angle* elements determine the geometry of a rectangular area. Separation indicates the distance between each pass. The values given in the specification are interpreted as default values. During the flight they will be set accordingly to the values provided by the operator or a mission control service. When one of these parameters changes the flight path will be recomputed.

Listing 9. XML description of a parametric leg.

```
<leg id="missleg" xsi:type="BasicScanLeg">
   <!-- Origin, size and orientation parameters come first -->
   <origin>41°17'35"N 1°54'25"E</origin>
   <dim1>1500</dim1>
   <dim2>-1200</dim2>
   <angle>240</angle>
   <separation>400</separation>
</leg>
```

Eventually a library of different parametric legs will be available complete enough so that a wide range of missions can be performed. With the use of parametric legs two goals are achieved. First, complex trajectories can be generated with no need to specify a possibly quite long list of legs. Second, the UAV path can dynamically adapt according to the input values.

III.F. Conditions

There are several points in the flight plan where conditions can be found: namely in holding patterns, iterative and intersection legs. For intersection legs, they are necessary in order to determine what path to follow next. For the rest of legs they will let the FPMS know when to leave the current leg and proceed to the next one.

Conditions will not be directly specified in the flight plan. Instead, each leg that depends on a condition will contain a reference which will be used to identify the condition. Conditions will be stored and processed separately. When the outcome of a condition is set the Flight Plan Manager Service (see next section) will be notified so that this change is taken into account for waypoint generation.

Conditions can be based on elapsed flight time, whether some task has been completed, on counters and on operator input. Operator input will always override any automatically generated outcome. In case of conflict or if the condition cannot be resolved we will resort to the default value or await user input.

IV. Flight Plan Manager

The previous section described the formalism used for specifying flight plans. This section presents the service responsible for processing and executing them: the Flight Plan Manager Service (FPMS). The FPMS forms part of a wider ecosystem of services that, together, provide the UAS with all its capabilities. On-board services can be classified into four groups: payload operation, mission services, awareness services

and flight services. The FPMS belongs to the latter category, it will collaborate with other on-board and remote services in order to execute the given flight plan. These interactions are described in section IV.A. To complete the description of the FPMS, section IV.B gives some details on its internal workings.

IV.A. Flight Plan Manager Service

The FPMS is responsible for executing flight plans, but it doesn't operate in a stand-alone fashion. To achieve its goals it collaborates with other services. The main service the FPMS collaborates with is the Virtual Autopilot Service (VAS), which is the only service with direct access to the installed autopilot (see figure 6). Apart from the VAS, the most relevant services the FPMS interacts with are the Ground Control Station (GCS) and the Mission Control Service (MCS). The GCS provides monitoring and control capabilities to a human operator. The Mission Control Service is in charge of evaluating conditions and updating parametric legs. This section describes the functionality offered by the FPMS. It is organized considering who is the main user of each part. We start with the VAS, followed by the GCS and the MCS.

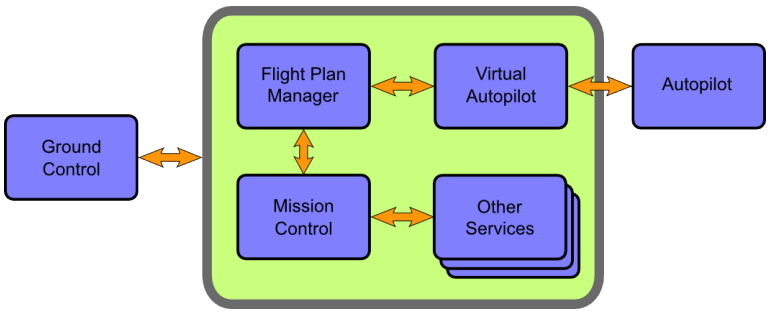


Figure 6. Flight Plan Manager Service interactions.

IV.A.1. Virtual Autopilot Service

The VAS operates similarly as drivers work on an operating system abstracting away autopilot implementation details from its users. In our system it is the only service with access to the on-board autopilot. Its main purpose is to make the UAS independent of any particular autopilot solution. To this end, it implements an autopilot-independent interface that provides telemetry data in a standardized form, supports a number of navigation commands and notifies about autopilot problems via status and alarms messages.

In order to execute the flight plan the FPMS will send navigation commands to the VAS. These commands mainly consist in waypoints the aircraft has to fly to. Since the flight plan is specified in terms of legs some translation process is needed for converting them into the waypoint sequences expected by the VAS. Computing waypoint sequences that approximate the legs specified in the flight plan is the main task of the FPMS execution engine. This flow of waypoint commands is the main form of interaction between the FPMS and the VAS. Sometimes, these waypoints will be accompanied by additional fields indicating speed and altitude change requests.

Other commands related to waypoint management include clearing all sent but pending waypoints. This will be necessary, for instance, when an emergency which forces to execute an alternative plan occurs. The FPMS will also issue partial cancellation commands as would be the case when ignoring a number of waypoints in order to directly jump to a given leg. Another type of command will allow the FPMS to change the VAS operation mode to request special operations such as taking-off or landing.

The information that will flow from the VAS to the FPMS will basically consist in notifications about the VAS state and flown waypoints.

IV.A.2. Ground Control Station

The Ground Control Station is the interface to the system that human operators will interact with. As such it must be able to manage and control several aspects of the flight plan and its execution. To this end, the FPMS provides the operations listed in table 2.

Apart from the listed operations, the FPMS also provides a number of information flows that enable monitoring. This data consists in the position of the aircraft in flight plan terms, i.e. what are the current

Table 2. Operations available to GCS

Load flight plan	Load the flight plan. Before starting any mission a flight plan must be submitted to the UAS.
Set initial leg	Since each stage of the flight plan can store multiple paths, when there is more than one possibility for the first flight plan stage, the GCS operator will indicate which one to start with.
Start	Initiate aircraft flight. An automatic or manual take-off, depending on UAS capabilities and configuration will be performed.
Pause	The aircraft will fly a holding pattern until commanded to resume flight plan execution.
Manual	Inform the FPMS that we are going into manual mode.
Resume	Switch from paused or manual mode to normal automatic operation.
Stop	Stops FPMS operation.
Goto leg	Fly directly to the given leg skipping intermediate ones without abandoning automatic mode of operation.
Update flight plan	Update command provided for modifying the flight plan. A waypoint can be moved, the values of a parametric leg can be updated and other leg parameters such as speed and altitude can be changed.
Set condition result	Set the result of any of the conditions the flight plan depends on.
Trigger emergency return	The FPMS will switch to the emergency flight plan defined for the current leg, stage or flight plan.

stage, leg and other leg-related information such as current iteration of an iterative leg, etc. It also provides information about the current operating state of the service.

IV.A.3. Mission Control Service

One of the main features of our flight plan specification mechanism is that it enables the UAS to adapt to mission circumstances. This can be done in two manners: first, with the possibility of using conditions in different leg types. Second, by using parametric legs, whose final form depends on the values of the input parameters.

Conditions mark a point where a decision can be made about what path to follow next. The decision-making process is not directly done by the FPMS. The flight plan contains references to condition identifiers. The outcome of these conditions will be set by the Mission Control Service. The MCS will also decide what are the actual parameter values for parametric legs. These functions are currently done from the GCS but the inclusion of the MCS will provide the UAS with a high degree of automation.

Other services of the USAL that will eventually make it into the UAS are those related to collision avoidance and Air Traffic Control interaction. The first ones imply that, during a period of time, some other service will take control and change the aircraft trajectory. The second ones introduce an external element whose commands will also affect the FPMS. The FPMS will need to be notified of these situations and, when normal operation can be resumed, perform a recovery operation if necessary.

IV.B. Flight Plan Execution

The main task of the FPMS consists in generating the sequence of waypoints that will direct the UAS flight. At the same time the FPMS has to respond to commands sent from the GCS and be aware of situations where VAS control is taken over by other services. This results in the FPMS operating in a number of states as seen in figure 7. A description of each one of these stages follows.

- On Command: The FPMS can be either on command or on standby. If on command it has control over the VAS and determines the path followed by the aircraft. When an emergency occurs the main

flight plan is replaced by the corresponding emergency plan and waypoint generation starts over. If the flight plan is updated all unflown waypoints affected by the change will be cancelled and replaced by new ones. *On Command* is a superstate that encompasses two substates, namely *Auto* and *Paused*.

- Auto: When in this state the FPMS is generating and forwarding waypoints to the VAS.
- Paused: The FPMS has received a pause command. This directly translates to issuing a change of state command to the VAS requiring it to perform a holding pattern. When the resume command is received it will notify the VAS that waypoint navigation can continue.
- On Standby: This state is entered when the FPMS is notified that some other service has taken control over the VAS. It can be entered because another service is trying to avoid a collision or because the VAS is now under manual control. It differs from the *Paused* state in that the FPMS is not going to send any message to the VAS. It just waits and tries to recover and continue flight execution once it regains control.

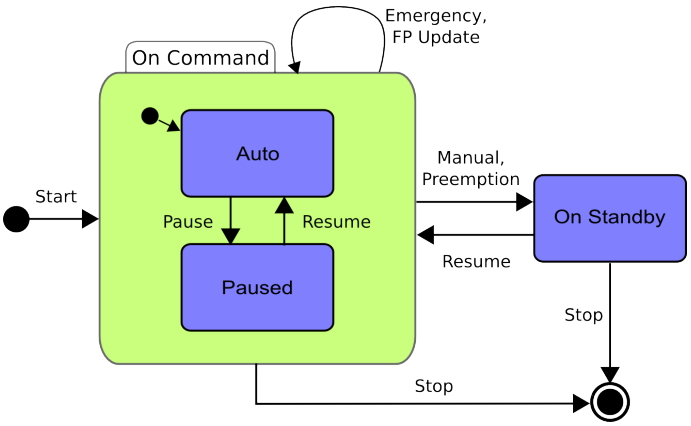


Figure 7. FPMS states.

Once the submitted flight plan has been loaded into the FPMS, an internal representation is generated and the service is ready to start waypoint generation. The flight plan is represented by a tree whose root node corresponds to the whole flight plan (see figure 8). Stages are located at the next level of the tree, legs follow. At this point some degree of recursion can be found due to iterative legs, whose children legs form the body of the iterative structure. Finally each leg has an associated waypoint. When the start command is received a traversal of the tree begins. The execution engine goes through each one of the flight plan stages, processing the legs they contain and generating waypoints as appropriate. Legs which present curved paths are approximated by sequences of waypoints.

Waypoint generation works in a decoupled manner from the FPMS operation depicted in figure 7. The FPMS is implemented following a producer-consumer model. There is one worker thread in charge of waypoint generation. Each new waypoint is stored in a queue. The consumer will pass the waypoints from the queue on to the VAS. The head of the queue contains the waypoint the VAS is heading to. The queue is also used to keep track of unsent waypoints. Using this scheme, only the consumer needs to be aware of petitions or changes happening in the system (a waypoint has been reached, the flight plan updated, manual mode entered, etc.) and command the generation engine to take appropriate action if necessary (e.g. restart at a given leg). When a condition is encountered the producer will block and no waypoints will be generated until the condition outcome is available. At this point the consumer will monitor the state of the queue and ensure that a decision is made in time, otherwise a default path, if present, will be taken. If the queue becomes empty (all waypoints have been flown), the VAS will be commanded to perform a holding pattern.

V. Simulation

In order to validate the proposed flight plan specification and execution mechanisms a Fight Plan Manager Service prototype has been developed. This prototype implements XML flight plan document processing,

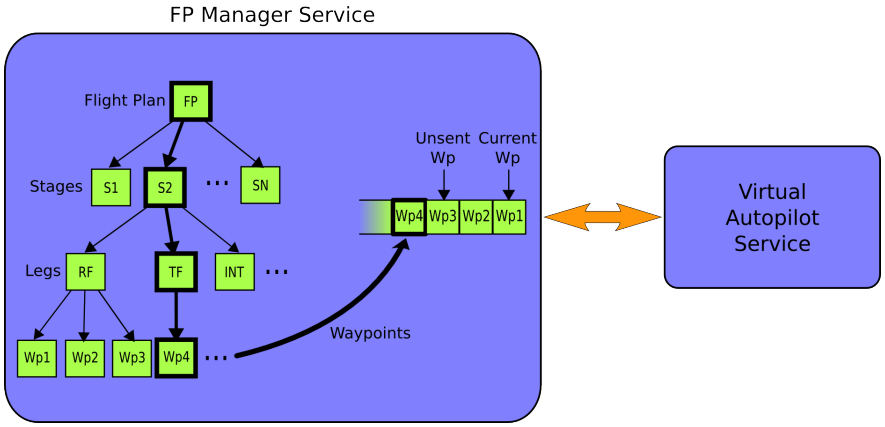


Figure 8. Flight Plan execution.

waypoint generation and simplified condition handling. Other USAL services, namely the Virtual Autopilot Service and the Ground Control Station are emulated using ad-hoc developments. The prototype also allows pre-flight waypoint generation for its validation. Next section presents three external applications used during the simulation (FlightGear Flight Simulator, Atlas and Google Earth) and their connection with our own tools. Then the simulated mission, consisting in monitoring the evolution of a hypothetical wildfire, is introduced and finally the simulation results are discussed.

V.A. Simulation Environment

To test the FPMS prototype without engaging in real flights a flight simulator is needed. Among several available options the FlightGear Flight Simulator (FGFS) has been chosen. It offers a wide range of input/output protocols enabling it to be easily controlled and monitored from an external program. Being multi-platform is also a nice feature. It's available at no cost and the fact of it being a free-libre open source project has simplified development: For instance, FlightGear doesn't allow to control aircraft navigation by means of arbitrary user defined waypoints, but replicating a small part of its built-in route manager this behaviour can be easily emulated. We have also been able to take advantage of some parts of code implemented in the simulator libraries thus reducing development costs.

Atlas is another free-libre open source project. It provides a map creator and a map viewer to produce and display charts of the world for users of FlightGear. It can be used to visualize flight evolution in a moving map display while the simulation is running. Since Atlas was designed to work together with FlightGear, connecting them is straightforward. Figure 9 shows at its right hand the connection between the two. FlightGear and Atlas communicate using a network socket.

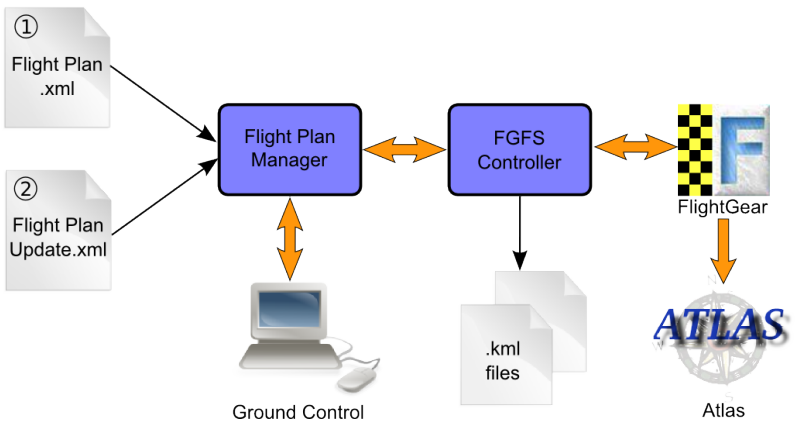


Figure 9. Simulation environment.

Figure 9 shows also the three developed services that complete the simulation environment: the FPMS,

a VAS-like service (FGFS Controller) and a console program acting as the GCS. The FPMS, which is the target of this validation experiment, can be found at the center of the figure. This service reads in the XML file specifying the flight plan, dynamically generates waypoints according to the input data and sends them to the autopilot through the VAS.

Since FlightGear is acting as our autopilot, we need an ad-hoc VAS development to interact with it. We refer to it as the FGFS Controller and has been implement as a thread running in parallel with the FPMS. The FGFS Controller connects with the FPMS using shared memory. Communication with FlightGear is done via sockets. The main task accomplished by the FGFS Controller consists in sending heading, speed and altitude parameters to control the simulated aircraft and reach the next FPMS generated waypoint. To compute these parameters it requires position data which it retrieves from FlightGear. The FGFS Controller also generates two KML files that enable post-processing of the flight data.

The simulation environment is completed with a simple console program, acting as the GCS, that provides some feedback and enables user interaction. All legs that make use of conditions in the flight plan will depend on user input to determine what to do next. Updates of the flight plan are also controlled by the user.

In our experiment we assume no contingencies, thus no services related to its management are included in the simulation. Once the flight plan is defined and the different tools are ready the simulation can start: the UAV takes off and flies to the mission area. Navigation is governed by a series of waypoints sent to the FGFS Controller by the FPMS. The FGFS Controller translates the received waypoints to autopilot instructions and sends a notification back each time a waypoint is reached. The GCS and Atlas show the flight evolution to the operator. The operator intervenes to set the result of flight plan conditions and to send flight plan updates to the FPMS.

An additional tool is used for post processing flight data: Google Earth has proved very user friendly and also very useful for checking waypoint generation and analyzing flight execution. The waypoints generated by the FPMS are stored in a KML file during the simulation and then visualized on a world map using Google Earth. The position data generated by FlightGear is also stored in the same manner, thus providing an excellent means for comparing generated waypoints against the aircraft trajectory. The FGFS Controller is responsible for generating the two aforementioned files: It receives waypoints from the FPMS and the position coordinates from FlightGear. Since this KML-formatted data can also be sent via an HTTP network link, Google Earth could easily become a replacement for Atlas in a near future.

Note that eventually both the GCS and the FGFS Controller will be replaced by their corresponding service implementations and their connection will be done using the middleware. These services will honour the USAL specification, in the case of the VAS, providing a device-independent interface to the payload on-board. Afterwards we will be able to replace the flight simulator by a flight control system unit only accessed by the VAS. This same scheme will be used for the rest of on-board devices, therefore the proposed architecture will also greatly simplify the simulation and testing of the system.

V.B. Experiment Description

Among the many possible applications of UAS to civil missions we are focusing on its use to support firefighters operations during and after wildfires. Our experiment will consist in the monitorization of a simulated fire in an area close to Barcelona. The input data will be generated by the popular wildfire simulator FARSITE.[?] The experiment starts with the information of the first five hours of fire growth. Using this information, an on-ground operator decides a take-off runway and designs a flight plan for exploring the fire area. Both, the burned area and the exploration area are shown in figure 10-A. For the specification of the area to explore the user just defines an origin vertex, the dimensions of the area and an angle indicating its orientation. The separation between each pass is also defined, as well as other extra parameters to define the geometry of a holding pattern. This holding pattern would be executed for turning if the separation didn't allow for a direct turn. A graphical flight plan editor, with GIS facilities and a library of pre-built procedures, would be used to automatically generate the corresponding XML document that describes the flight plan.

The central part of the flight plan is the mission stage. Listing 1 shows the use case of the mission stage for this experiment in its XML form. It consists in an iterative leg that repeatedly covers an area with a parametric leg: The iterative leg contains only one leg (labelled missleg, as in mission-leg) which is repeated up to 5 times; The missleg is a parametric leg with initial parameter values that approximate the fire area after the first five hours of burning.

Listing 10. Mission stage example

```
<!-- XML code corresponding to the mission stage -->
<stage id="mission" type="Mission">
   <name>Mission</name>
   <description>Perform mission</description>
   <legs>
      <!-- An iterative leg will cause the scan pattern to repeat -->
      <leg id="missloop" xsi:type="IterativeLeg">
         <body>missleg</body>        <!-- Contains only missleg    -->
         <first>missleg</first>      <!-- which is the first       -->
         <last>missleg</last>        <!-- and last leg of the loop -->
         <upperBound>5</upperBound> <!-- Repeat five times        -->
         <cond>loop_term</cond>      <!-- Termination condition    -->
      </leg>
      <!-- The basic scan pattern leg -->
      <leg id="missleg" xsi:type="BasicScanLeg">
         <!-- Origin, size, orientation and separation parameters come first -->
         <origin>41.2933169313151 1.907006250982991</origin>
         <dim1>5410</dim1>
         <dim2>-4200</dim2>
         <angle>322.5</angle>
         <separation>800</separation>
         <!-- The rest determine how teardrop turns would be done if needed -->
         <turndirection>Right</turndirection>
         <d1>700</d1>
         <d2>450</d2>
      </leg>
   </legs>
   <!-- Initial and final legs of this stage -->
   <initialLegs>missloop</initialLegs>
   <finalLegs>missloop</finalLegs>
</stage>
```

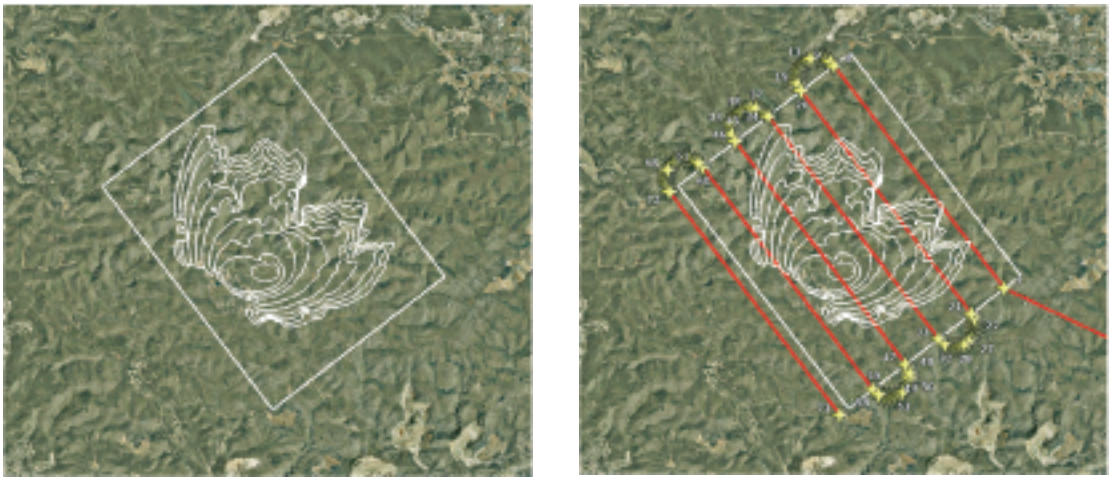
The iterative leg terminates when indicated by a human operator or after 5 iterations. Then the remaining flight plan legs are executed to route back and land in the designed runway. The parametric leg of this mission stage is a basic scan pattern. A scan pattern will cover totally the area of interest in parallel flight over it. The turning manoeuvres between each parallel track are done outside of the area of interest. The FPMS is responsible for the generation of the intermediate waypoints that make the UAS follow a scan pattern. Figure 10-B shows a global view of the intermediate waypoints needed to execute a basic scan pattern, while figure 10-C details the set of waypoints generated for a single turning manoeuvre.

The experiment continues with the UAS flying over the area while the fire evolves. After one or more scans the UAS operator will have a global view of the fire state and will update the scan pattern parameters in accordance with the fire evolution. At this point the FPMS will recompute the flight path to adapt to the new situation. Listing 2 shows the example of an XML modification command. This command looks very much as a small part of a flight plan. The <*change*> tag is used to inform the FPMS that some contents of the flight plan are going to be modified. In this case, new values are given to the parameters of the basic scan leg. The parametric leg is given a new origin, different dimensions and a new angle as shown in figure 10-D. The FPMS will use this data for a dynamic update of the flight plan.

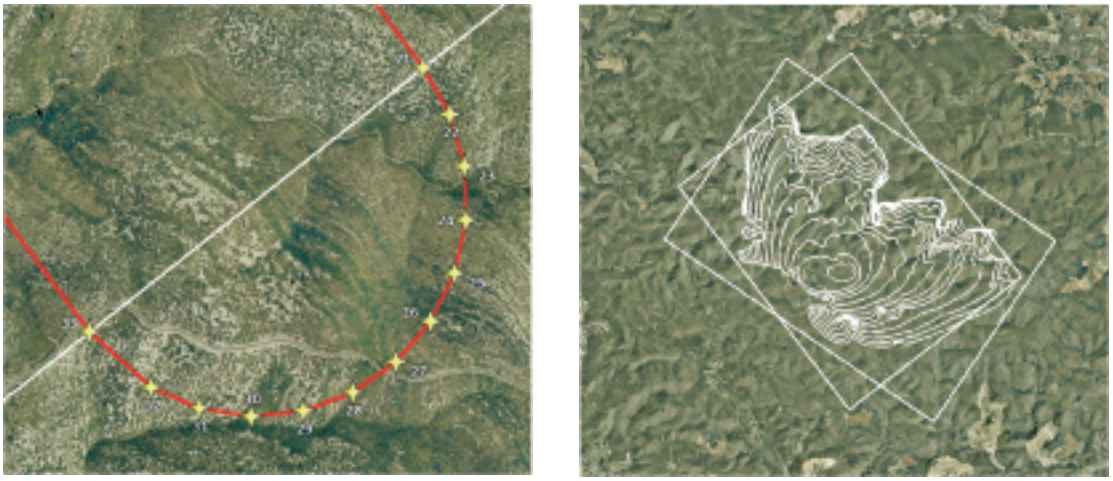
Eventually, mission control and image processing services will be available making human intervention unnecessary. An image processing service would be in charge of tracking fire evolution. The mission control service would then dynamically send the variations of the area of interest to the FPMS.

Listing 11. Update message with changes to scan pattern leg

```
<FlightPlan>
   <!-- First tag indicates that we are modifying an existing leg -->
   <change>
      <!-- Identify updated leg and its location within the flight plan -->
      <plan targetId="FireMission">
         <stage targetId="mission">
            <leg targetId="missleg">
               <!-- Set new values -->
               <origin>41.2995061043129 1.914776836073949</origin>
               <dim1>6275</dim1>
               <dim2>-4200</dim2>
               <angle>304</angle>
```



(A) (B)



(C) (D)

Figure 10. Waypoints generated.

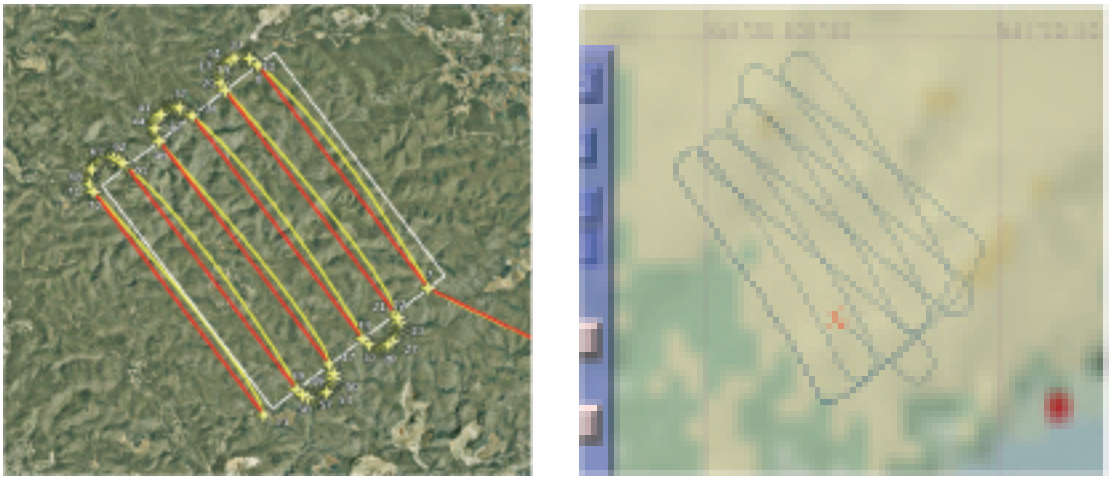
```
                <!-- Parameters not present are not modified -->
            </leg>
        </stage>
    </plan>
</change>
</FlightPlan>
```

V.C. Simulation Results

The purpose of the simulation is twofold: First of all, the validation of the flight plan specification mechanism as a way for describing flight plans; And second, test its flexibility for adapting to a changing environment. In figure 11-A we can observe the results of the simulation for the flight plan fragment displayed in figure 10-B. Red lines show the individual segments that connect the FPMS generated waypoints and overprinted in yellow colour, the actual flight of the UAV can be seen. The actual flight in our experiment is executed by the FlightGear Flight Simulator. We can observe that the prescribed flight is not exactly equal to the actual flight because of the simulated wind value, but we observe that the area is totally covered. Since coverage of the area is the main interest of a scan pattern the results are satisfactory.

To validate the flexibility of the flight plan specification we performed the simulated flight over a dynamically changing area of interest, as would be the case for a live fire. The results demonstrate how easily the

aircraft trajectory can dynamically be adapted by means of parametric legs. This mechanism is simple to use for an end user and it is also quite powerful in mission terms. Figure 10-D shows two areas that cover the fire spread with two hours of difference. In both cases the necessary information to specify the area of interest are its origin and dimensions plus the angle which defines its orientation.



(A) (B)

Figure 11. Updated scan area.

We can observe the final simulated flight in figure 11-B and check that the two areas of interest are being effectively covered during the mission time. This figure is a screen shot of the Atlas visualization taken after approximately two and a half hours of simulation.

VI. Conclusion

With current UAS airframes and FCS technology the construction of an unmanned aerial vehicle can be considered a relatively easy task. But there are still a number of important technical and non-technical issues that remain to be solved, e.g. airspace integration and legal matters. This article focuses on an additional aspect which we believe is key for the successful introduction of UAS into the civil market, i.e. the provision of UAS with the necessary tools for transforming them into flexible and highly capable systems for a wide variety of missions. This article tries to partially fill this mission level gap with the proposal of a novel flight plan specification. The proposed specification mechanism improves on the common list of waypoints approach by providing RNAV-like legs as the main unit for flight plan construction. Besides the leg concept is extended to include higher level control structures for specifying iterative behavior and branching. The decision making for this control structures can be based on mission variables, therefore enabling the UAS to respond depending on mission time information. The adaptivity of the system to mission time circumstances is also greatly improved by the inclusion of parametric legs. With parametric legs, the flight path of the aircraft is dynamically generated depending on its input parameters.

To manage the flight plan execution we propose the introduction of a UAS module we refer to as the Flight Plan Manager Service. The main task of this module consists in processing the flight plan and translate the leg based specification into suitable waypoint commands that will be sent to the FCS. To validate the concept a prototype of the flight plan manager has been implemented and its operation tested in a simulation environment.

As future work, there are some improvements that should be made to the mission manager. First aircraft performance needs to be taken into account when dynamically generating waypoints. Also, the dynamic behavior of the system makes it difficult to fully predict the flight path of the aircraft beforehand. Although eventually other services of the architecture will be in charge of avoiding collision, the flight manager should be extended with planning capabilities for generating safe paths. Finally a major effort has to be put in extending the flight plan management capabilities presented in this paper with payload control. To this end, we envision an evolution of the presented flight plan manager collaborating with a mission control service that will be able to dynamically change the aircraft behavior as more appropriate to fulfill the mission goals.

Acknowledgments

This work has been partially funded by Ministry of Science and Education of Spain under contract CICYT TIN 2007-63927.

The work presented in this paper has been performed with support from the Innovative Studies Programme of the EUROCONTROL Experimental Centre[a].

References

[1]Haiyang, C., Yongcan, C., and YangQuan, C., "Autopilots for Small Fixed-Wing Unmanned Air Vehicles: A Survey," *International Conference on Mechatronics and Automation (ICMA)*, IEEE, Harbin, China, 2007, pp. 3144–3149.

[2]"Aeronautical Information Manual, Official Guide to Basic Flight Information and ATC Procedures," U.S. Federal Aviation Administration, 2007.

[3]"Guidance Material for the Design of Terminal Procedures for Area Navigation," European Organisation for the Safety of Air Navigation, 2003.

[4]Giglio, L., Descloitres, J., Justice, C., and Kaufman, Y., "An enhanced contextual fire detection algorithm for MODIS," *Photogrammetric Engineering and Remote Sensing*, Vol. 87, 2003, pp. 273–282.

[5]Wegener, V. A. S. and Brass, J., "The UAV Western States Fire Mission: Concepts, Plans and Developmental Advancements," *AIAA 3rd "Unmanned Unlimited" Technical Conference, Workshop and Exhibit*, AIAA, Chicago, Illinois, 2004.

[6]Casbeer, D., Kingston, D., Beard, R., and McLain, T., "Cooperative forest fire surveillance using a team of small unmanned air vehicles," *International Journal of Systems Science*, Vol. 37, No. 6, 2006, pp. 351–360, Publisher: Taylor and Francis Ltd.

[7]Pastor, E., Lopez, J., and Royo, P., "UAV Payload and Mission Control Hardware/Software Architecture," *IEEE Aerospace and Electronic Systems Magazine*, Vol. 22, No. 6, 2007.

[8]Lopez, J., Royo, P., Pastor, E., Barrado, C., and Santamaria, E., "A Middleware Architecture for Unmanned Aircraft Avionics," *ACM/IFIP/USEUNIX 8th Int. Middleware Conference*, NewPort, California, Nov. 2007.

[9]W3C, "W3C Note: Web Services Architecture," http://www.w3c.org/TR/ws-arch.

[10]"UPnP Forum," http://www.upnp.org.

[11]Santamaria, E., Royo, P., Lopez, J., Barrado, C., Pastor, E., and Prats, X., "Increasing UAV Capabilities Through Autopilot and Flight Plan Abstraction," *Proceedings of the 26th Digital Avionics Systems Conference*, Dallas, Texas, 2007.

[a]EUROCONTROL Innovative Studies: http://www.eurocontrol.int/eec/public/standard_page/WP_Innovative_Studies.html