An Event Driven Approach for Increasing UAS Mission Automation

Eduard Santamaria∗, Cristina Barrado†, Enric Pastor†
Technical University of Catalonia, Barcelona, Spain

UAS have great potential to be used in a wide variety of civil applications such as environmental applications, emergency situations, surveillance tasks and more. 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.

Despite the increasing number of COTS components becoming available, much effort is still required in order to make UAS a viable commercial solution for civil applications. We believe that for UAS to be successful in this context they must be flexible systems able to perform a wide variety of missions with minimal reconfiguration and reduced operational costs.

In previous work, a flight plan specification formalism and its corresponding execution engine have been presented. These elements may suffice for simple applications but for more complex scenarios, we need a mechanism that specifies the vehicle behavior not only in flight plan terms but also taking into account payload operation. To provide this integration and, at the same time, increase the level of automation a mission management layer is added on top of the flight plan management facilities. The system flexibility requirement is satisfied by decoupling the mission description from its execution engine. This paper introduces an XML based mission specification mechanism for modeling the event-driven state-based behavior of the UAS. The Mission Manager is the software module responsible for its execution. The integration of the Mission Manager with other components that form part of our UAS distributed architecture is also described.

Nomenclature

FCS Flight Control System
FGFS FlightGear Flight Simulator
FPM Flight Plan Manager
MMa Mission Manager
RNAV Area Navigation
SCXML State Chart XML
UAS Unmanned Aircraft System
UAV Unmanned Aerial Vehicle
VAS Virtual Autopilot System
XML Extensible Markup Language

I. Introduction

Unmanned Aircraft Systems (UAS) offer great promise for an increasing amount of civil applications. With the consolidation of Flight Control System (FCS) technologies and the incorporation of cheap Commercial Off-The-Shelf (COTS) electronics, the usage of UAS is now extending to the civil market. However, much effort is still required to make UAS a viable commercial solution for civil applications. We believe

∗Assistant Professor, Computer Architecture Dept., Avda. del Canal Olimpic 15, 08860 Castelldefels, Spain.
†Associate Professor, Computer Architecture Dept., Avda. del Canal Olimpic 15, 08860 Castelldefels, Spain.
that for UAS to be successful in this context they must be flexible systems able to perform a wide variety of missions with minimal reconfiguration and reduced operational costs. An increased autonomy level will reduce operators workload, further improve cost-effectiveness and provide other benefits like a more efficient use of available bandwidth.

This paper presents advanced mission capabilities that integrate UAS flight plan and payload operation and provide a higher level of automation while still satisfying the flexibility requirements of the system. An XML based mission description is used to model the event-driven state-based behavior of the UAS. This description is processed and executed by the Mission Manager (MMa), which orchestrates payload operation and interacts with the Flight Plan Manager (FPM) to adapt the aircraft flight to the mission needs.

Being a key component of the system, the FPM is also described to some extent. The FPM is responsible for executing flight plans, which are also represented by means of an XML document. Our flight plan specification formalism offers semantically much richer constructs than those present in most current UAS autopilots, which rely on simple lists of waypoints. The FPM interacts with the on-board FCS to direct the aircraft according to the prescribed flight plan.

However, the flight plan alone does not suffice for specifying a complete mission, where payload operation needs to be considered in order to achieve the mission goals. Three elements are identified as necessary to fully specify a mission:

1. The tasks to undertake.
2. The way to perform the tasks.
3. Any temporal constraints that may exist between the tasks or behaviors.

An example mission used throughout the paper consists in monitoring a burned area in search of hotspots. This mission is composed of separate tasks, such as patrolling a particular area or analyzing potential hotspots. These tasks are broken down into the individual actions or behaviors that must be undertaken to achieve them. A temporal constraint is imposed by the fact that a potential hotspot must be found before taking a closer look at it.

Different Finite State Machine (FSM) variants have traditionally been used in the robotics field to specify behavior. The Witas project presents an UAS related example where extended FSM diagrams are used to facilitate the specification of tasks. Dong et al. also make use of FSMs in their proposed architecture for UAS control. Barbier et al. follow a similar approach but making use of a different formalism. In this case Petri Nets are used instead.

In our system State Chart XML (SCXML) is used to model the UAS behavior. SCXML is a working draft published by the World Wide Web Consortium that provides a general purpose event-driven state machine language based on Harel’s Statecharts. Two important aspects that differentiate statecharts from traditional state machine languages are their ability to express modularity and concurrency, thus making them more appropriate for the description of complex systems. It is worth mentioning that RSML, a language based on statecharts, has been used in past years to formalize system requirements of critical avionics systems.

The organization of the paper is as follows: Section II presents the network centric architecture of the system. Section III describes the flight plan specification formalism. Section IV provides an overview of statecharts and SCXML. In section V the integration of the MMa and the FPM is discussed. Section VI presents the simulation environment and the simulation results of our hotspots mission example. Finally, section VII concludes the article and identifies some future developments.

II. System Overview

This section describes the UAS architecture, a distributed embedded system on board the aircraft 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 layer for managing communications.

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 distributed and scalable architecture. This approach offers a simple scheme with a number of
benefits in our application domain that motivate its selection:

- Development simplicity: 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.

- Flexibility: 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.

- Easy node interconnection: In contrast with the complex interconnection schemes needed by end-to-end parallel buses, with a LAN based approach new nodes can be added with much less effort.

This architecture accommodates a number of software components that are organized following a Service Oriented Architectures (SOA). 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, that carries out some task to the benefit of a service consumer. This architecture provides enhanced interoperability, flexibility and extensibility of the designed system and of their individual 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 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.B. UAV Service Abstraction Layer

The UAV Service Abstraction Layer (USAL)\textsuperscript{17} is a set of services running on top of the middleware that give support to most types of remote sensing UAV missions (see Figure 1). The USAL can be compared to hardware drivers in 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. The USAL treats sensors and in general all payload in a similar manner.

The USAL defines a collection of reusable services that comprises a minimum common set of elements that are needed in most UAV missions. 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.

Available services have been classified into four categories:

1. Flight Services: all services in charge of basic UAS flight operations, including interaction with the autopilot, power system and engine monitoring, contingency management, etc.

2. Mission Services: all services in charge of developing the actual UAS mission, e.g. mission management and real time data processing.

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 to terrain avoidance and integration into non-segregated airspace.
There are three USAL services which are of special relevance to the work presented in this paper, namely the Mission Manager (MMa), the Flight Plan Manager (FPM) and the Virtual Autopilot System (VAS). The FPM and the VAS belong to the Flight Services category, while the MMa belongs to the Mission Services category. As shown in Figure 2, there is a hierarchical relationship between them. The VAS is the closest to the on-board flight control system and it has two main objectives: (1) provide waypoint based navigation abstracting autopilot details from the rest of the system, and (2) extract internal sensor information from the autopilot and offer it to other services for its exploitation during the UAS mission. The FPM stands between the VAS and the MMa. It provides flight plan management capabilities for executing a leg-based flight plan. Finally, the MMa interacts with the FPM updating leg parameters and conditions contained in the flight plan to adapt the UAS trajectory to the needs of the ongoing mission.

**Virtual Autopilot System** The VAS directly interacts with the installed autopilot abstracting away the autopilot implementation details from its users. It also offers a number of information flows, such as UAS position, attitude, etc. to be exploited by other UAS services. 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. From the navigation point of view, the VAS provides waypoint based navigation plus a number of states to handle the different flight phases and contingency situations.

**Flight Plan Manager** The flight planning capabilities of most autopilots are 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. The FPM has been developed to implement much richer flight-plan capabilities on top of the functionality offered by the autopilot. The FPM enables structured flight plan definition using RNAV-inspired 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. 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 are much simpler than those available in the flight plan manager, the FPM translates the flight plan legs into a sequence of waypoints to be processed by the VAS. The VAS dynamically feeds the autopilot with these waypoints during mission time, therefore transforming the FPM into 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 FPM, we obtain a highly capable platform.

**Mission Manager** The MMa orchestrates payload operation and interacts with the FPM to adapt the aircraft flight to the mission needs. An incremental approach has been followed in the design of the UAS architecture so that the MMa is not strictly required for operating the system. However, with its addition the UAS benefits from a much improved level of autonomy. The UAS behavior is modeled using a state based control scheme based on Harel’s statecharts. This model is stored in an XML document which the MMa takes as its initial input. Events received by the MMa drive the execution of the model which, in turn, generates other events targeting UAS services. This interchange of information enables the MMa to effectively control the operation of the different UAS components in a coordinated manner.

### III. Flight Plan Specification Formalism

Previous section has introduced the UAS architecture. In this section we present the formalism that is used for describing flight plans. Most current UAS autopilot systems rely on lists of waypoints as the mechanism for flight plan specification and execution. This is a very restrictive approach: it is difficult to specify complex trajectories, changes to the flight plan may imply having to deal with a considerable amount of waypoints, there is no support for conditional or iterative behavior and it does not facilitate reuse of flight plan fragments. In short, current autopilots specialize in low level flight control and navigation is limited to very basic go to waypoint commands. For these reasons a new flight plan specification mechanism has been proposed that provides higher level constructs, with richer semantics, and which enables adaption to mission progress.

Some ideas are based on current practices in commercial aviation industry for the specification of Area Navigation (RNAV) procedures. 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, which describe a path (e.g., heading, course, track, etc.) and a termination point (e.g., the path terminates at an altitude, distance, fix, etc.). Our specification mechanism borrows some leg types from RNAV and extends them with additional constructs. New control constructs such as iterative legs and intersection legs are added and adaptivity is increased by means of parametric legs. Further details are given in the next subsections, which describe the flight plan structure and its elements.

#### III.A. Flight plan

The flight plan is represented by means of an XML document that contains the navigation instructions for the UAS. This document contains the main flight plan plus a number of alternatives for emergency situations. Each one of them is composed of stages, legs and waypoints hierarchically organized as seen in Figure 3.

Stages are the largest building blocks within a flight plan. They organize legs into different phases that will be performed in sequence. Legs specify the path that the plane must follow in order to reach a destination waypoint. Several primitives for leg specification are available. A waypoint is a geographical position defined in terms of latitude/longitude coordinates. A waypoint may also be accompanied by target altitude and speed indications. Optionally, a partial flight plan to be carried out if an emergency occurs can be associated to a flight plan. This emergency plan will be superseded by emergency plans specified at stage or leg level. A partial flight plan follows the same structure as the main flight plan but contains only those stages necessary to fly from the current position to the landing runway of choice.
III.B. Stages

Stages constitute high-level building blocks for flight plan specification and are used to group together legs that seek a common purpose. They correspond to flight phases that will be sequentially executed. Table 1 lists currently available stage types.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taxi</td>
<td>Move to or return from runway.</td>
</tr>
<tr>
<td>TakeOff</td>
<td>Description of take off operations.</td>
</tr>
<tr>
<td>Departure</td>
<td>Legs flown after take off to reach the starting point of the next stage.</td>
</tr>
<tr>
<td>EnRoute</td>
<td>Cruise to a destination area.</td>
</tr>
<tr>
<td>Mission</td>
<td>Series of legs that will be flown during main mission operations.</td>
</tr>
<tr>
<td>Arrival</td>
<td>Legs connecting the end of the route with the approach procedures.</td>
</tr>
<tr>
<td>Approach</td>
<td>Prepare for landing.</td>
</tr>
<tr>
<td>Land</td>
<td>Landing operation.</td>
</tr>
</tbody>
</table>

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 stage 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 departure procedures that start at different positions depending on the chosen take-off direction. 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 that enable the flight plan designer to provide this one-to-one correspondence.

III.C. 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.C.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 scanning 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.C.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: 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: Corresponds to a straight trajectory from waypoint to waypoint. The initial position is the destination waypoint of the previous leg.

• Direct to a Fix: 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: 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 which differ in how they are terminated. Hold to an Altitude terminates when a given altitude is reached. 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 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 is no longer satisfied (regardless of the number of iterations).

III.C.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 4 shows the two different possibilities for iterative leg specification. Diagram 4a 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 4b shows a simpler case when the legs of the body are executed one after another in a single direction.

Figure 4: Iterative leg types.
III.C.3. Intersection Legs

Intersection legs are used in situations where there is more than one possible path to follow and a decision needs to be made (see figure 5). This leg type contains a list with the different alternatives and a condition for picking one of them. Intersection legs are also used to explicitly indicate where two or more different paths meet.

Together with parametric and iterative legs, intersection legs provide a powerful means for adapting the flight as best suited to the ongoing mission circumstances.

![Figure 5: Intersection legs.](image)

III.C.4. Parametric Legs

With parametric legs complex trajectories can be automatically generated from a reduced number of input parameters. If the actual values of these parameters change, the resulting trajectory will be dynamically recomputed. In this way, the aircraft trajectory can be modified depending on the evolution of mission variables. Figure 6 shows a number of possible patterns for exploring a given area, as in (a) and (b), or a more specific point, as in (c) that can be obtained using parametric legs.

![Figure 6: Scanning patterns.](image)

Eventually a complete enough library of different parametric legs will be available 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 to the mission requirements.

III.D. Conditions

There are several points in the flight plan where conditions can be found: namely in holding patterns, iterative legs 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 FPM know when to leave the current leg and proceed to the next one.

Conditions are not directly specified in the flight plan. Instead, each leg that depends on a condition contains an identifier, which is used to refer to the condition, and sometimes also a default value. Conditions are processed separately and when a condition is given a result, i.e. an integer value telling which is the selected path, the FPM dynamically recomputes affected waypoints.

IV. Mission Modeling

In the previous section the flight plan specification has been described. This specification enables the flight plan designer to create plans that can be submitted to the UAS and executed by the Flight Plan Manager in collaboration with the Virtual Autopilot System. While these components alone already provide
a very capable platform, the inclusion of a mission control layer on top of them will greatly improve the autonomy of the system. Again, we aim at obtaining a flexible system able to adapt to different mission scenarios. Therefore, the Mission Manager, which is going to be in charge of the mission execution, needs to be as generic as possible. For that reason the MMa is conceived as an execution engine for mission descriptions stored in XML format.

For the description of the UAS mission we propose the use of State Chart XML (SCXML). SCXML is a working draft published by the World Wide Web Consortium, it provides a general purpose event-driven state machine language based on Harel’s statecharts. Statecharts can be used to model the behavior of a system by means of a finite number of states, transitions between those states, and actions. Statecharts extend traditional state machine diagrams with support for hierarchy, concurrency and communication. They enable modular descriptions of complex systems, provide constructs for expressing concurrency and a broadcast mechanism for communication between concurrent components. Statecharts are part of the Unified Modeling Language (UML), which is a widespread graphical modeling language used in industry and academia.

IV.A. State Diagrams

A state diagram describes the behavior of a system in terms of states, events and transitions. A state reflects the current configuration of the system. A trigger that causes a transition to occur is called an event. A transition is a relationship between two states. It indicates that a system in the first state will enter the second state when a specified event occurs and the specified guard conditions are satisfied. Conditions are boolean expressions used to specify under what circumstances a given transition is permitted. Finally, transitions can be also accompanied by actions. An action specifies an executable computation that takes a very short amount of time to complete. Typical things actions are used for include causing another event to fire, updating some data structure and interact with the outside world. Fast execution times are required because ideally transitions should occur instantaneously.

Graphically, a state diagram is a collection of nodes representing states and arcs representing transitions. Each transition has a label that comes in three parts: event [guard]/action. All three parts are optional.

As an example, Figure 7 shows a state diagram for a simple reconnaissance mission. The diagram depicts the different states the UAS would go through in order to complete the mission. The filled circle indicates the initial state. The first state that will be entered is OnGround. When an event is received notifying that the UAS is taking off the transition to TakeOff takes place. The system remains in this state until the enroute event is received. Note that although there is a clear mapping between flight plan stages and mission states this relationship doesn’t need to be one to one. From the EnRoute state two different destinations can be reached: Mission and Landing. The flight plan imposes a sequential order between the EnRoute, Mission and Arrival stages. When the Mission stage is entered the system will transition to the Mission state, when done it will return to EnRoute. Finally the Landing state encompasses all arrival and landing procedures specified in the flight plan. Once on ground the system goes back to the initial state. Some actions could be added to the diagram in Figure 7. For instance, during the transition to the Mission state some mission specific payload could be turned on. It could later be turned off when abandoning this state.

Figure 7: Statechart for a simple reconnaissance mission.

Harel’s statecharts extend traditional state machines diagrams with support for hierarchy and concurrency. Hierarchical decomposition enables the mission designer to structure the behavior specification in
smaller and more manageable pieces. The concurrency capabilities of statecharts can dramatically reduce the complexity produced by state explosion found in traditional state diagrams, where all state combinations need to be considered. In Section V we will take advantage of hierarchical decomposition to refine the Mission state. A number of substates will be used for a more detailed specification of the UAS behavior during this phase. Section V also includes an example where concurrency is used.

IV.B. StateChart XML (SCXML)

SCXML is a state machine language being developed by the World Wide Web Consortium (W3C). A Working Draft\textsuperscript{11} has been published which defines a complete specification to represent statechart diagrams. SCXML is organized in different modules, the most relevant ones are listed below:

- Core Module: contains the elements that define the basic Harel state machine. It provides elements to specify states, transitions and some executable content, e.g. raising an internal event. The set of elements which can appear as executable content is extended by other modules.
- External Communications Module: adds the capability of sending and receiving events from external entities, as well as invoking external services.
- Data Module: implements the capability of storing, reading and modifying a set of data that is internal to the state machine.

Each one of these modules defines a set of tags and its semantics. Listing 1 shows the SCXML representation of the statechart depicted in Figure 7. The first line is the XML declaration. It defines the XML version (1.0) and the character encoding (UTF-8) the document makes use of. The next line describes the root element, in this case it says that this is an SCXML document. An attribute in the root element indicates the initial state. Afterwards the five states of our example mission can be found, each one with its own id. Each state contains the list of transitions with origin at that state. Taking the transition inside the OnGround state as an example, we found that it contains three attributes: event, cond and target. Event indicates which event triggers the transition, in this case the event is called current\_stage. Cond specifies a condition, we are going to abandon the OnGround state and enter the transition target only if the current flight plan stage is takeoff. Note that this differs from Figure 7 which offers a simplified view. Most of the events present there really translate to a single current\_stage event plus a condition. Finally, target indicates the destination state of the transition. Another point worth noting is the src attribute in the Mission state. The value of this attribute is the name of a file containing a submachine that refines this state.

Listing 1: SCXML encoding of UAS mission state diagram.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0" initialstate="OnGround">
  <state id="OnGround">
    <transition event="current\_stage" cond="eventdata=='takeoff'" target="TakeOff"/>
  </state>

  <state id="TakeOff">
    <transition event="current\_stage" cond="eventdata=='goroute'" target="EnRoute"/>
  </state>

  <state id="EnRoute">
    <transition event="current\_stage" cond="eventdata=='mission'" target="Mission"/>
    <transition event="current\_stage" cond="eventdata=='arrival'" target="Landing"/>
  </state>

  <state id="Mission" src="MissionStateA.xml">
    <transition event="current\_stage" cond="eventdata=='reroute'" target="EnRoute"/>
    <transition event="terminate" target="EnRoute"/>
  </state>

  <state id="Landing">
    <transition event="on\_ground" target="OnGround"/>
  </state>
</scxml>
```
All elements appearing in Listing 1 belong to the Core Module. More advanced examples appear in Section V showing how to take advantage of elements from the External Communications and Data Modules to fully specify the UAS behavior.

V. Flight Plan and Mission Integration

In previous sections the formalisms used to specify the flight plan and the UAS behavior have been separately introduced. In this section we describe how they can be used in conjunction to carry out an example mission. Before that, for a better understanding of how the two pieces work together the interaction between the MMa and other services, with a main focus on the FPM, is described. The MMa and the FPM services are respectively responsible for executing the flight plan and mission specifications.

The example used to illustrate the flight plan and mission integration consists in the detection and analysis of potential hotspots once a wildfire has been suppressed. This task is necessary to make sure that no remaining hotspots can restart the fire. The use of UAS in this scenario can be very beneficial to firefighters because nowadays a significant number of resources must be allocated to the burned area to prevent a fire restart from happening.

To carry out the mission the UAS will fly a scanning pattern over a rectangular area of interest. This operation will allow the system to detect potential hotspots that will be later on analyzed performing an eight pattern over each one of them. Once all potential hotspots have been visited, the UAS will execute a holding pattern awaiting further instructions. To carry out this mission we need camera and sensor related services for inspecting the ground surface, data processing services to analyze the acquired data, storage services, etc. We assume that all these are available. Each time a potential hotspot is detected the MMa will receive a notification together with its coordinates. In our simplified example, which concentrates on the mission and flight plan integration, this is the only needed information.

Two cases are presented that illustrate how the example mission can be performed in different ways using the same flight plan specification. Version A assumes that some time consuming processing is required before realizing a potential hotspot should be analyzed in more detail. In this case we will first complete the scanning operation and, afterwards, visit the potential hotspots one by one. In version B we assume that more capable embarked services are able to detect potential hotspots immediately. Taking advantage of that, the UAS will fly the eight pattern upon hotspot detection and then resume the scanning of the area, thus exhibiting more advanced flight control capabilities. Figure 7 shows the main states the UAS goes through to complete the mission. This statechart is common to both versions. The differences between version A and B appear when the Mission state gets refined.

Next section describes the interaction between the MMa and the FPM, i.e. what kind of message interchange takes place between these (and other) services. Afterwards the flight plan used to perform the example mission is presented. Finally, the UAS behavior specifications for the two mission cases are discussed. The simulation results obtained with the two approaches are presented in Section VI.

V.A. MMa and FPM Interaction

Figure 8 shows the message exchange between the MMa and the FPM. Coordinated operation requires the MMa to continuously listen to the FPM and keep track of the flight evolution. At the same time, the MMa must be able to generate the necessary control messages to make the FPM adapt the flight trajectory to the current mission situation. The MMa also interacts with other services to receive mission related information (e.g. a potential hotspot has been detected) and control payload operation (e.g. activate a given sensor when the Mission state is entered). Table 2 summarizes the main messages involved in this process.

One of the advantages of using a representation which may eventually become an standard, as is the case for SCXML, is that we can benefit from already existing tools. As displayed in Figure 8, our MMa prototype makes use of Commons SCXML, a library from the Apache Software Foundation that provides a generic event-driven state machine based execution environment, borrowing the semantics defined by SCXML. The MMa implements a bridge which enables communications between the UAS services and the execution engine provided by Commons SCXML. The MMa translates messages coming from the FPM, e.g. a notification that execution of a certain leg has started, to SCXML events. Symmetrically, messages sent by the execution engine, e.g. setting a new result value for a given condition, are also handled by the MMa.

In this section we have covered only those messages that are effectively used in our example mission. The
Figure 8: The MMa bridges communications between UAS services and the SCXML execution engine.

<table>
<thead>
<tr>
<th>Direction</th>
<th>Message</th>
<th>Parameters</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPM → MMa</td>
<td>currentLeg</td>
<td>leg id</td>
<td>Execution of a new flight plan leg has started.</td>
</tr>
<tr>
<td></td>
<td>currentStage</td>
<td>stage id</td>
<td>Idem for flight plan stages.</td>
</tr>
<tr>
<td>MMa → FPM</td>
<td>setCurrentCondition</td>
<td>cond id, value</td>
<td>Set the result value of a flight plan condition.</td>
</tr>
<tr>
<td></td>
<td>fpUpdate</td>
<td>update string</td>
<td>Modify the parameters of one or more legs. Update string contains modification instructions in XML form.</td>
</tr>
<tr>
<td></td>
<td>gotoLeg</td>
<td>target leg id</td>
<td>Jump directly to the specified leg.</td>
</tr>
<tr>
<td>VAS → MMa</td>
<td>position</td>
<td>lon, lat, alt</td>
<td>UAS position.</td>
</tr>
<tr>
<td>other → MMa</td>
<td>hotspot</td>
<td>lon, lat</td>
<td>Potential hotspot location.</td>
</tr>
</tbody>
</table>

MMa will be able to interact with any service on the UAS network in a similar manner, the only restriction being that the appropriate translations must be provided. Another way to think about it is considering the MMa as a wrapper that handles all communications between the SCXML engine and the outside world. As such it determines the view of the system as seen by the SCXML engine.

V.B. Flight Plan

The flight plan to perform the hotspots mission consists in a sequence of stages that take the UAS platform to the mission area, followed by the Mission stage and, finally, another sequence of stages to return to base. Most stages contain one or more Track to a Fix legs that describe the path for going to or returning from the mission area. The interesting part of the flight plan corresponds to the Mission stage, where the constructs that enable the adaption to the mission needs can be found. Figure 9 shows an schematic view of the Mission stage, which includes three different flight patterns used for the following purposes:

- **scanArea**: Leg that follows a flight pattern that systematically scans a rectangular area.
- **scanPoint**: Fly an eight pattern over a point of interest. In our example mission it is used for analyzing a potential hotspot.
- **hold**: Hold to a Fix leg used in conjunction with missLoop to wait until commanded on what to do next.

An intersection leg (patternSelect) is placed before these legs. The intersection leg contains a condition whose result governs the selection of the next leg. If the result of the condition is 0 scanArea is selected, scanPoint is selected if it is 1 and the holding pattern if it is 2. A second intersection leg (join) is placed after the three pattern legs to explicitly indicate that there is a single path from that point on. An iterative leg (missLoop) is used in order to repeat this selection and be able to alternate between the different leg types. Also note that both the intersection and the iterative legs are control structures that do not determine a flight path by themselves.
Figure 9: Organization of the legs contained in the Mission stage of the flight plan.

Listing 2 shows the XML code corresponding to the Mission stage of the flight plan. For brevity some parameters have been omitted. As it can be seen, the stage element contains four parts: a descriptive name, a list of legs and two elements indicating which are the initial and final legs of the stage. The list of legs contains the five legs that appear in Figure 9: missLoop, patternSelect, scanArea, scanPoint, hold and join. Except for missLoop itself, all these legs form the body of the iterative leg, which is both the first and last leg of the stage. PatternSelect contains a condition named selection used to pick one of the three possible next legs. By default the scanArea leg is selected. After patternSelect the three legs providing the different flight patterns are found. Each of them depends on a number of parameters that determine the aircraft trajectory. All of them have join, which is the last leg of the loop body, as next leg.

Listing 2: Specification of the flight plan’s mission stage for hotspots detection.

```
<stage id="mission" type="Mission">
  <name>Hotspot detection mission</name>
  <legs>
    <leg id="missLoop" xsi:type="IterativeLeg">
      <body>patternSelect scanArea scanPoint hold join</body>
      <first>patternSelect</first>
      <last>join</last>
      <upperBound>15</upperBound>
    </leg>
    <leg id="patternSelect" xsi:type="IntersectionLeg">
      <next>scanArea</next> <!-- default value -->
      <nextCond>selection</nextCond>
      <nextList>scanArea scanPoint hold</nextList>
    </leg>
    <leg id="scanArea" xsi:type="BasicScanLeg">
      <next>join</next>
      <origin>41.58875930014136 1.690274338399033</origin>
      <dim1>6000</dim1>
      <dim2>5500</dim2>
      <angle>80</angle>
      <separation>800</separation>
      <startAt>41.58875930014136 1.690274338399033</startAt>
    </leg>
    <leg id="scanPoint" xsi:type="ScanPointLeg">
      <dest>
        <coordinates>41.56947331267459 1.717810982215079</coordinates>
      </dest>
      <next>join</next>
      ...
    </leg>
    <leg id="hold" xsi:type="HFLeg">
      <next>join</next>
      ...
    </leg>
    <leg id="join" xsi:type="IntersectionLeg" />
  </legs>
  <initialLegs>missLoop</initialLegs>
  <finalLegs>missLoop</finalLegs>
```
The MMa controls the flight path in several ways. It can use the `fpUpdate` message to set the parameters of the leg that is going to be flown. For example, if a scan point pattern is to be flown, the MMa will update the corresponding leg with the actual coordinates of the point of interest. To do so the accompanying parameter of the `fpUpdate` message contains a string whose content looks like in Listing 3. A `change` tag is used to indicate we want to modify a value. Then a number of elements are used to identify the leg we want to change and, finally, the new value for the specified parameter is given.

Listing 3: Update message to set new coordinates of the scanPoint leg.

```xml
<FlightPlan xmlns="">
  <change>
    <plan targetId="HotSpotPlan">
      <stage targetId="mission">
        <leg targetId="scanPoint">
          <dest>
            <coordinates>41.58020309593712 1.736978105734669</coordinates>
          </dest>
        </leg>
      </stage>
    </plan>
  </change>
</FlightPlan>
```

With the `setCondition` message the MMa has control over what is going to be flown. Other means of interaction include the `gotoLeg` message and other commands not covered in this paper.

V.C. Mission (version A)

In this section we discuss the so-called version A of the mission implementation. In version A we assume that some time is required for analyzing the collected samples and decide that a given location should be analyzed in more detail. In other words, we need some time to realize that the UAS has passed upon a potential hotspot. The expected behavior for this version of the mission is to fully scan a rectangular area first and, afterwards, visit each one of the potential hotspots.

Figure 10 shows a refinement of the Mission state found in Figure 7. The substates the Mission state is decomposed into are distributed between two parallel regions. A dashed line in the figure separates both regions. When the Mission state is reached two parallel substates are simultaneously entered: HotSpotsCounter, which is used to keep track of the number of encountered potential hotspots, and ScanArea, which performs a scan over the area of interest. There are a number of actions which are not reflected in the statechart but are coded in the SCXML document. We are going to discuss what is going on during the Mission state first, and show an example of some of the involved SCXML code afterwards.

The operation of the HotSpotCounter state is as follows: each time a hotspot event is delivered a counter is incremented by one. When this happens we are certain that there is at least one potential hotspot that needs to be visited. Therefore, during the HotSpotsCounter’s self-transition we also set the coordinates of the scanPoint leg to the first non-visited potential hotspot and modify the selection condition in `patternSelect` so that scanPoint is picked.

In the parallel region found at the bottom of Figure 10 there are three states which directly map to the corresponding flight plan legs. The ScanArea is the mission state the system remains in when the scanArea leg is executed and the same relationship is established between the Hold and ScanPoint states and their leg counterparts. Transitions between states are triggered by the FPM making progress. If we are in the ScanArea state and, at some point, the scanPoint leg starts its execution the FPM will notify the MMa and this will trigger a transition from the ScanArea state to the ScanPoint state. To adapt the flight plan to the mission needs we follow these steps:

1. First, upon entering a state, we set the result of the selection condition to control which leg is going to be flown next. This can be thought of as setting a default next leg.

2. Then, if some event is received while in the current state that requires the next leg decision to be reconsidered, we make use of the transition triggered by the event to make the necessary updates to the flight plan and change the result of the selection condition.
Figure 10: State chart for version A of the Mission state: hotspot analysis is deferred.

For example, when entering ScanArea we set the selection condition to 2, meaning that hold is going to be our default leg. If there is a hotspot that needs to be explored the HotSpotCounter self-transition is triggered and the following actions are performed:

1. The scanPoint leg is updated with the coordinates of the first unvisited potential hotspot.
2. The result of the selection condition in patternSelect is set to 1 to select scanPoint as next leg.

When the scanArea leg ends, if no hotspots have been detected, the system will enter the Hold state. Otherwise, execution of the scanPoint leg will start and we will transition to the ScanPoint state. The system will remain in the ScanPoint state until all potential hotspots have been visited. In this state, the selection of the default leg depends on the number of remaining unvisited hotspots. Finally the system will enter the Hold state. When in the Hold state, if a scan event is received, the flight plan will be updated so that the whole scanning process is started over.

To give an idea of how this behavior is translated to an SCXML document, Listing 4 shows the encoding of the ScanPoint state. The different assign operations included in this listing operate against the data elements defined as shown in Listing 5 (discussed below). The first thing done on entering this state is incrementing the number of visited hotspots to reflect the current execution of the scanPoint leg. Next we set the default next leg. If there are unvisited hotspots we are going to set scanPoint as the next leg. This requires the scanPoint target to be set. We rely on an external object (an object provided by the MMA but not directly contained in the document) to store and access the detected potential hotspots. With this data an update message is composed that gets sent to the FPM. Because our system relies on a subscription based communications infrastructure managed by a middleware layer called MAREA, we set target and targettype attributes of the send action respectively to container and x-marea. This is a convention used to indicate that the middleware container the MMA runs inside will be used to deliver the message to all subscribers by means of the protocols defined by MAREA. The selection condition of patternSelect is not updated because it already points to the scanPoint leg. The else branch is taken when there are no pending hotspots. In this case we set hold as the default next leg.

Listing 4: SCXML encoding of ScanPoint state.

```
<state id="ScanPoint">
<onEntry>
  <!-- Increment the number of visited hotspots -->
  <assign name="visit_hs_count" expr="visit_hs_count + 1" />
  <!-- Selection of next leg depends on the number of pending hotspots -->
  <if cond="visit_hs_count lt detect_hs_count">
    <!-- Set scanPoint target to first non-visited potential hotspot -->
    <assign name="lat" expr="HotSpotList[visit_hs_count].getLatitude().toString()" />
  </if>
</onEntry>
```
Listing 5 shows the data elements used in the previous SCXML code. Most of them are temporary variables that do not maintain any kind of state information. This is not the case for detectHotspotsCount and visitHotspotsCount which respectively store the number of detected potential hotspots and the number of visited ones. The scanPointUpdate data element is of special interest because it provides an skeleton of the XML code sent for updating the flight plan (compare it to the example message of Listing 3). This data element is accessed using the Data() function, a proper value is set to the coordinates field and the result is sent as a parameter of an fpUpdate message.

Listing 5: Data elements supporting ScanPoint state operations.

V.D. Mission (version B)

In this section we present an alternative version of the mission implementation that assumes that a potential hotspot can be detected as soon as it is approached. When such detection takes place, we expect the system to change its trajectory and perform an eight pattern over the point of interest. After that, the UAS should resume the scan of the area where it was left.

In version B the Mission state of Figure 7 is refined as shown in Figure 11. The Mission state is decomposed into three different substates, all three of them having a direct mapping to the three legs implementing the different flight patterns. The initial state is ScanArea. When it is entered the UAS is
performing a scan over the rectangular area of interest. We follow the same philosophy as in version A. The first thing we do on entering a state is setting a default next leg by changing the selection condition of patternSelect. On entering ScanArea the selection condition is set to 2 (hold). This means that when the current leg finishes the next leg to be flown will be the holding pattern. There are three transitions with origin at ScanArea which need to be considered:

- **hotspot**: A potential hotspot has been detected. Instead of directly jumping to the ScanPoint state, we update the flight plan and wait for the eight pattern to start its execution. During this transition the MMAs does the following:
  1. Update the `startAt` parameter of the scanArea leg with the current position so that later on it can be resumed from there.
  2. Update the scanPoint leg with the coordinates of the potential hotspot that must be analyzed.
  3. Set the result of the selection condition to 1, i.e. select scanPoint as the next leg.
  4. Send a command to the FPM to skip the rest of the current scan and directly jump to the scanPoint. The response event generated by the FPM triggers the transition to the ScanPoint state.

- **scanPoint**: This transition does not require any special action.

- **hold**: If nothing happened during the scan, the flight will continue with a holding pattern. During this transition the flight plan is updated to ensure that if a new scan is necessary it will start from the beginning of the area of interest.

When the ScanPoint state is entered we set scanArea as the default next leg. When done with the eight pattern the UAS resumes the scanning of the area and seamlessly transitions to the ScanArea state.

![State Chart](image)

Figure 11: State chart for version B of the Mission state: hotspots are analyzed immediately.

Listing 6 shows the different actions taking place during the ScanArea self transition triggered by the hotspot event. Aircraft position data, is provided by the MMAs. The location of a potential hotspot, comes as a parameter of the corresponding event and is accessed using `_eventdata`.

Listing 6: ScanArea self transition for hotspot event.

```xml
<transition event="hotspot">
  <!-- (1) Update startAt parameter of scan so that later on it can be resumed -->
  <assign name="lat" expr="Position.getLatitude().toString()" />
  <assign name="lon" expr="Position.getLongitude().toString()" />
  <assign name="coordinates" expr="lat.concat(\'\').concat(lon)" />
  <!-- Compose update message -->
  <assign location="Data(scanUpdate, 'FlightPlan/change/plan/stage/leg/startAt')"
      expr="coordinates" />
  <!-- Send message to fpm -->
  <send target="container" targettype="x-marea"
      event="fpUpdate" namelist="scanUpdate" />

  <!-- (2) Update scanPoint leg with hotspot position data -->
  <assign location="Data(scanPointUpdate, 'FlightPlan/change/plan/stage/leg/dest/coordinates')"
      expr="_eventdata[\'coordinates\']" />
  <!-- Send message to fpm -->
  <send target="container" targettype="x-marea"
```
VI. Simulation

In section V two possible ways for performing a hotspot detection mission have been described. In this section we present the simulation environment that has been used to test both approaches and the obtained results.

VI.A. Simulation Environment

Our simulation environment is composed of the services and other programs depicted in Figure 12. Boxes above the network bar represent embarked components. Boxes below the network bar belong to the ground segment.

![Figure 12: Simulation environment.](image)

The first service found at the top left corner of figure Figure 12 is the VAS. The VAS provides waypoint navigation capabilities and a number of telemetry flows regarding the UAS position, attitude, autopilot status, etc. A very important feature of the VAS is that it isolates the real autopilot from the rest of the system, thus enabling the construction of systems that do not depend on a particular autopilot solution. To simulate the aircraft behavior and obtain flight data to feed the system FlightGear Flight Simulator is used.

Next to the VAS lies the FPM. The FPM receives a flight plan from the Ground Control station. It creates an internal representation of the plan and uses it to dynamically generate a sequence of waypoints. These waypoints feed the VAS. When condition results or flight plan updates are received, the FPM will recompute all affected waypoints. If invalidated waypoints have already been sent to the VAS, a message is sent informing that they must be discarded.

The last service on top of the network bar is the MMA. A prototype has been implemented that reads in the SCXML based specification of the mission and makes use of Commons SCXML for its execution. In its current form this prototype only listens to the aircraft position (generated by the VAS) and to events notifying that execution of a given leg has started (coming from the FPM). The MMA is implemented as a wrapper around the SCXML engine that bridges communications with the the rest of the system.

Although in this paper we focus mainly in its interaction with the FPM, the same mechanisms enable the MMA to communicate with any other service. In this paper, we assume the existence of the necessary payload...
to take images, analyze them and decide whether further inspection of a given point is required. Besides tracking flight progress, the detection of a potential hotspot is the only event the execution engine needs to be aware of. This event must be accompanied by the hotspot coordinates. For a clearer comparison of versions A and B of the example mission the same hotspots are used in both cases. The MMA stores the hotspots’ data and feeds the engine with the hotspot events according to the mission assumptions. This approach also satisfies the storage capabilities required to implement version A. In version A it is not enough to know the amount of detected hotspots, its deferred analysis requires their positions to be stored somewhere.

Below the network bar GoogleEarth and a Ground Control station can be found. The Ground Control station consists in a number of consoles that enable interaction with the embarked services. It is used, among other things, to submit the flight plan description.

To keep track of the mission evolution we make use of GoogleEarth. This virtual globe application continually queries the UAS position to a web server connected to the flight simulator and shows its position in real time.

VI.B. Simulation Results

Using the described simulation environment the proposed approach for increasing UAS mission automation has been tested. The FPM has been initialized with the flight plan described in Section V.B. Two simulations have been performed, one for version A and one for version B of the example mission, with three potential hotspots at fixed positions. Figure 13 shows the obtained results. To facilitate the comparison, the results from versions A and B have been placed side by side in two different columns. The first row shows the UAS trajectory halfway through the mission. The second row displays the trajectory of a complete mission execution.

![Figure 13: Results obtained with both versions of the example mission.](image-url)
Focusing on version A, as it can be seen in Figures 13a and 13c, it is only when the whole area has been scanned that the analysis of potential hotspots starts. After getting to the end of the scan, the aircraft turns left and proceeds to the first hotspot. It performs an eight pattern at that location and then continues with the remaining ones. Once an eight pattern has been flown above each one of them the UAS waits performing a holding pattern until commanded to return.

In version B (Figure 13b) the aircraft changes its path to perform an eight pattern as soon as it approaches the first hotspot. As expected, once the eight pattern has been flown, it is able to resume the scan and do the same for the two remaining hotspots (see also 13d).

Comparing both versions we can see how our system is able to perform the same mission in different ways by changing the mission specification. This level of flexibility enables the user to specify the UAS behavior as best fit considering payload capabilities and other requirements. It is noteworthy that the deferred hotspot analysis of version B requires their locations to be memorized. Another important difference is that for an optimal traversal of the potential hotspots planning capabilities are also required.

VII. Conclusion

This paper presents the Mission Manager service as a new fundamental part of the network centric architecture proposed for UAS. In our approach the UAS is a network that connects several loosely decoupled software components known as services. In previous work we presented the Flight Plan Manager service as responsible for executing the UAS flight plan. The definition of intersection and parametric legs in the flight plan enables the UAS to exhibit dynamic behavior. The Mission Manager presented in this paper is mainly responsible for dynamically taking decisions about the flight execution and communicating them to the Flight Plan Manager using the network.

The flight plan is the central focus of the mission execution. The different flight plan phases are represented in the Mission Manager as mission states. Events from the flight, the payload or the UAS operator may produce mission state transitions. A W3C state machine formalism has been used to define and implement the core of the Mission Manager service. This service subscribes to some of the events produced by other services and forwards them to the state machine executor. The Mission Manager also publishes the outputs of the state machine execution to the rest of the services, for example to communicate the modification of a leg parameter to the Flight Plan Manager. It can also provide additional capabilities like storing generated information, as when maintaining a list of potential hotspots for its later exploration.

The benefit of this flexible architecture is that it enables UAS services reuse across diverse civil missions. The surveillance of hotspots after a wild fire is used as the motivating example of its potential. Using the same flight plan, two mission approaches are shown: first with a complete execution of each flown leg and, afterwards, with the interruption and later resumption of the scanArea leg. The state chart of both examples and the XML listings of some transitions show that the mission specification can be accomplished with ease.

Future work includes improvements in the schedule to visit each potential hotspot in version A of the example, the completion of other related services as the Image Processing or the Mission Monitor, and the extension of the Mission Manager for other missions like surveillance of the fire front of a live fire. The State Machine approach for the implementation of a fundamental service has shown its potential for the implementation of other services of the UAS.

Acknowledgments

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

This work has been co-financed by the European Organisation for the Safety of Air Navigation (EUROCONTROL) under its CARE INO III programme. The content of the work does not necessarily reflect the official position of EUROCONTROL on the matter.

References


