Formal Mission Specification and Execution Mechanisms for Unmanned Aircraft Systems

EDUARD SANTAMARIA BARNADAS
Computer Engineer

Advisers
DR. ENRIC PASTOR LLORENÇ
DR. CRISTINA BARRADO MUXI

Department of Computer Architecture
Technical University of Catalonia

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya (UPC)

A dissertation submitted for the degree of
Doctor of Philosophy
April 2010

Formal Mission Specification and Execution Mechanisms for Unmanned Aircraft Systems

Technical University of Catalonia
April 2010

PhD. Thesis made in:
Department of Computer Architecture
Castelldefels School of Technology (EPSC)
Esteve Terradas, 7.
08860 Castelldefels
Catalonia (Spain)

A l'Astrid, la Bruna i en Marçal.

Contents

List of Figures

List of Tables

List of Publications

The list of publications resulting from this PhD. work is given in inverse chronological order as follows:

Conference Proceedings

- Santamaria, Eduard, Barrado, Cristina, & Pastor, Enric. 2009 (Apr.). An event driven approach for increasing UAS mission automation. *In: Proceedings of the AIAA Unmanned...Unlimited 2009 Conference*. AIAA, Seattle, Washington (USA).
- Santamaria, Eduard, Pérez, Marc, Ramírez, Jorge, Barrado, Cristina, & Pastor, Enric. 2009 (Sept.). Mission formalism for UAS based navaid flight inspections. *In: Proceedings of the 9th AIAA Aviation Technology, Integration, and Operations (ATIO) conference*. AIAA, Hilton Head, South Carolina (USA).
- Santamaria, Eduard, Royo, Pablo, Barrado, Cristina, & Pastor, Enric. 2009 (Aug.). An integrated mission management system for UAS civil applications. *In: Proceedings of the AIAA Guidance, Navigation, and Control 2009 Conference*. AIAA, Chicago, Illinois (USA).
- Santamaria, Eduard, Royo, Pablo, Barrado, Cristina, Pastor, Enric, López, Juan, & Prats, Xavier. 2008 (Aug.). Mission aware flight planning for unmanned aerial systems. *In: Proceedings of the AIAA Guidance, Navigation, and Control 2008 Conference*. AIAA, Honolulu, Hawaii (USA).
- Santamaria, Eduard, Royo, Pablo, López, Juan, Barrado, Cristina, Pastor, Enric, & Prats, Xavier. 2007 (Oct.). Increasing UAV capabilities through autopilot and flight plan abstraction. *In: Proceedings of the 26th Digital Avionics Systems Conference*. IEEE/AIAA, Dallas, Texas (USA).

Agraïments

Thanks to the following institutions for partially funding this work: Ministerio de Educación (under contract CICYT TIN 2007-63927) and EUROCONTROL Experimental Centre (through its Innovative Studies Programme).

La meva arribada a l'Escola Politècnica Superior de Castelldefels es va produir a finals del 2004. En aquells moments, vaig començar a conèixer els meus nous companys i a observar, amb interés, com un petit grup iniciava una aventura en un camp totalment nou per a mi: els vehicles aeris no tripulats. Al capdavant d'aquest grup hi havia l'Enric Pastor. No va passar gaire temps fins que em vaig apropar a l'Enric per expressar-li el meu desig d'incorporar-me al seu equip. La seva resposta va ser afirmativa, i el meu primer agraïment, doncs, va per l'Enric. A l'Enric li haig de donar les gràcies per haver tingut la valentia d'embarcar-se en aquest projecte, per haver-me permès formar-ne part, per haver-me proposat un tema de recerca interessant i estimulant i per tot el suport que des de llavors m'ha donat.

Als voltants d'aquella època, hi va haver una altra persona que també va apostar per formar part d'allò que s'estava gestant i que acabaria sent el grup ICARUS. Aquesta persona és la Cristina Barrado. La Cristina ha tingut un paper clau en aquesta tesi i li haig d'agraïr les moltes hores que m'ha dedicat. Amb ella he discutit molts detalls d'aquest treball i ha estat una revisora i consellera incansable. També li vull agraïr molt especialment el seu suport, sobretot en els moments difícils, que també n'hi ha hagut.

En Xavier Prats, en Pablo Royo i en Juan López completen el grup de persones que em vaig trobar en aquells inicis. A en Xevi, li haig d'agraïr la seva part de culpa en què un dia l'Enric decidís dedicar-se als vehicles aeris no tripulats. Gràcies, també, per tots els coneixements que ens has aportat des de la vessant aeronàutica i que han tingut un impacte important en aquest treball. I gràcies per la plantilla i l'ajuda amb el LaTeX. En Pablo i en Juan són els responsables de les altres peces del puzzle del qual aquesta tesi forma part. A ells, els dono les gràcies per l'excel·lent feina que han realitzat.

Els darrers anys, el grup de recerca ha anat creixent amb la incorporació de cares noves. Una persona que mereix una menció especial és en Marc Pérez, sempre disposat a ajudar. També gràcies a en Jorge Ramírez, per les col·laboracions que hem establert en diferents moments, i a tota la resta de gent del grup.

No vull oblidar-me d'en Noel Trillo que, en el seu treball de fi de carrera, va contribuir a la implementació d'algunes parts del codi fent una feina excepcional. Donar gràcies, també, a l'Helena Flores pel seu cop de mà amb l'anglès.

La meva arribada a Castelldefels va significar també el començament de la meva activitat docent. En aquest àmbit he après molt de la mà d'en Miguel Valero i de la resta de companys. Són molts i no els anomenaré tots, però a tots ells gràcies. Gràcies pel plaer que és treballar amb vosaltres, per tot el que he après i per la comprensió i complicitat ajudant-me a poder dedicar el màxim temps possible a tirar endavant aquesta tesi. Gràcies també a l'Eva Rodríguez, per l'excel·lent companyia durant el temps que vàrem compartir despatx.

Gràcies a tots els habituals (Dolors, Roc, Javier, Esunly, Esther, . . .), i als no tan habituals, de les tertúlies de l'hora de dinar. Des dels temes més tècnics fins a les discussions més surrealistes tot i té cabuda. Llàstima que, d'un temps ençà, em sigui més complicat afegir-me a les escapades al cinema o al teatre.

Vull dedicar els meus darrers agraïments a la meva família. Als meus pares i al meu germà, per haver fet que arribar fins aquí fos possible. Als sogres, cunyats, cunyades i les recents incorporacions, per la seva paciència i comprensió. Gràcies a la meva dona, l'Astrid, pel seu suport incondicional i per compartir al meu costat tots els moments bons i no tan bons. I finalment, gràcies a la meva filla Bruna i al meu fill Marçal que, sense entendre res de tot això, han aconseguit que fos molt més fàcil.

Castelldefels, Abril de 2010
Eduard Santamaria Barnadas

Resum

En els darrers temps estem assistint a un important creixement de l'interès en els vehicles aeris no tripulats, en anglès Unmanned Aircraft Systems (UAS), motivat per la constatació de la gran quantitat de possibles aplicacions d'aquest tipus de sistemes en l'àmbit civil. Aquests aparells poden ser de gran utilitat en aplicacions mediambientals, situacions d'emergència, operacions de vigilància i com a repetidors de comunicacions, entre altres. En general, són especialment indicats per a operacions que impliquen repetitivitat, perillositat o que s'han de portar a terme en entorns hostils.

La majoria de solucions comercials existents utilitzen sistemes de control de vol basats en navegació per *waypoints*, és a dir, l'aparell segueix la ruta indicada per una sèrie de punts a partir de les seves coordenades. Això, quan no es tracta de sistemes que són pilotats remotament. A més, la capacitat de coordinar l'operativa dels sistemes embarcats amb la fase del vol és inexistent. Per tant, les seves capacitats pel que fa a automatització i autonomia són molt limitades. Alguns elements motivadors per avançar cap a l'assoliment de més autonomia vénen donats per les limitacions en l'ample de banda, les limitacions en la capacitat d'atenció dels operadors humans durant períodes perllongats, un més ràpid accés a les lectures dels sensors i millor capacitat de resposta, així com l'abaratiment de costos que pot comportar una reducció en la càrrega de treball dels operadors i en l'entrenament necessari.

Altres requeriments que considerem clau per a l'èxit dels UAS en l'àmbit civil són les possibilitats de reconfiguració que ofereixin així com una limitació en els seus costos. Amb tot, hauríem d'obtenir plataformes assequibles capaces d'operar en diferents situacions amb poca intervenció per part d'operadors humans.

Per augmentar les capacitats dels UAS i satisfer els anteriors requeriments, proposem afegir capes de gestió del vol i de la missió per sobre dels sistemes de control de vol disponibles comercialment. D'aquesta manera, podrem aconseguir alts nivells d'autonomia tot traient profit de tecnologies ja existents i, en conseqüència evitant la necessitat de grans inversions. La capacitat de reconfiguració del sistema s'obtindrà separant l'especificació del vol i la missió dels elements encarregats de portar-ne a terme la seva execució.

Els components de gestió del vol i de la missió presentats en aquesta tesi s'integren en una més àmplia arquitectura *hardware/software* que està essent desenvolupada pel grup de recerca ICARUS. Aquesta arquitectura segueix un model basat en serveis on els subsistemes de l'UAS s'interconnecten mitjançant una infraestructura de xarxa comuna. Diferents components poden ser inclosos o extrets de la xarxa en funció de les necessitats de la missió que es vulgui dur a terme.

La primera contribució d'aquesta tesi consisteix en un nou llenguatge per a l'especificació del vol que permet descriure el vol en segments. Aquests segments permeten descriure no només el punt de destí, sinó també la trajectòria per arribar-hi, i per tant proporcionen un nivell d'abstracció superior al que ofereix una sistema basat exclusivament en *waypoints*. Aquest concepte s'amplia afegint-hi construccions que permeten especificar bifurcacions, comportament repetitiu i generació de trajectòries complexes a partir d'un nombre de paràmetres reduït.

També s'ha desenvolupat el servei de gestió del pla de vol (Flight Plan Manager o FPM), que es responsabilitza de la seva execució. Com que el sistema de control de vol subjacent segueix basant-se en *waypoints*, es generen, de forma automàtica, punts intermitjos per tal d'ajustar el vol a la trajectòria desitjada.

Per tal de coordinar el vol amb l'operativa de la càrrega útil també s'ha desenvolupat el servei de gestió de la missió (Mission Manager o MMa). El gestor de la missió pot ajustar el funcionament dels elements de càrrega útil d'acord amb quina sigui la fase actual del vol. De forma anàloga, té la capacitat d'actuar sobre el FPM i modificar el pla de vol segons les necessitats de la missió. Per especificar el comportament de l'UAS, en lloc de dissenyar un nou llenguatge, proposem la utilització de State Chart XML, un futur estàndard per l'especificació de màquines d'estat actualment en fase d'elaboració.

Finalment s'ha portat a terme la validació dels diferents elements d'especificació i execució mitjançant l'execució simulada de dos missions d'exemple. La primera missió imita els procediments necessaris per a la inspecció de ràdio ajudes i mostra el comportament de l'UAS durant l'execució d'un vol complex. En aquesta missió només s'ha utilitzat el FPM. El segon exemple combina el FPM i el MMa per dur a terme una missió que consisteix en la detecció de punts calents en una àrea determinada després d'un hipotètic foc forestal. En aquesta simulació es pot veure com el MMa és capaç de modificar el pla de vol per tal d'adaptar la trajectòria a les necessitats de la missió. En particular, es vola un patró en forma de vuit sobre cadascun dels potencials punts calents detectats dinàmicament.

Abstract

Unmanned Aircraft Systems (UAS) are rapidly gaining attention due to the increasing potential of their applications in the civil domain. UAS can provide great value performing environmental applications, during emergency situations, as monitoring and surveillance tools, and operating as communication relays among other uses. In general, they are specially well suited for the so-called D-cube operations (Dirty, Dull or Dangerous).

Most current commercial solutions, if not remotely piloted, rely on waypoint based flight control systems for their navigation and are unable to coordinate UAS flight with payload operation. Therefore, automation capabilities and the ability for the system to operate in an autonomous manner are very limited. Some motivators that turn autonomy into an important requirement include limited bandwidth, limits on long-term attention spans of human operators, faster access to sensed data, which also results in better reaction times, as well as benefits derived from reducing operators workload and training requirements.

Other important requirements we believe are key to the success of UAS in the civil domain are reconfigurability and cost-effectiveness. As a result, an affordable platform should be able to operate in different application scenarios with reduced human intervention.

To increase capabilities of UAS and satisfy the aforementioned requirements, we propose adding flight plan and mission management layers on top of a commercial-off-the-shelf flight control system. By doing so, a high level of autonomy can be achieved while taking advantage of available technologies and avoiding huge investments. Reconfiguration is made possible by separating flight and mission execution from its specification.

The flight and mission management components presented in this thesis integrate into a wider hardware/software architecture being developed by the ICARUS research group. This architecture follows a service oriented approach where UAS subsystems are connected together through a common networking infrastructure. Components can be added and removed from the network in order to adapt the system to the target mission.

The first contribution of this thesis consists, then, in a flight specification language that enables the description of the flight plan in terms of legs. Legs provide a higher level of abstraction compared to plain waypoints since they not only specify a destination but also the trajectory that should be followed to reach it. This leg concept is extended with additional constructs that enable specification of alternative routes, repetition and generation of complex trajectories from a reduced number of parameters.

A Flight Plan Manager (FPM) service has been developed that is responsible for the execution of the flight plan. Since the underlying flight control system is still waypoint based, additional intermediate waypoints are automatically generated to adjust the flight to the desired trajectory.

In order to coordinate UAS flight and payload operation a Mission Manager (MMa) service has also been developed. The MMa is able to adapt payload operation according to the current flight phase, but it can also act on the FPM and make modifications on the flight plan for a better adaption to the mission needs. To specify UAS behavior, instead of designing a new language, we propose using an in-development standard for the specification of state machines called State Chart XML.

Finally, validation of the proposed specification and execution elements is carried out with two example missions executed in a simulation environment. The first mission mimics the procedures required for inspecting navigation aids and shows the UAS performance in a complex flight scenario. In this mission only the FPM is involved. The second example combines operation of the FPM with the MMa. In this case the mission consists in the detection of hot spots on a given area after a hypothetical wildfire. This second simulation shows how the MMa is able to modify the flight plan in order to adapt the trajectory to the mission needs. In particular, an eight pattern is flown over each of the dynamically detected potential hot spots.

List of Acronyms

ADS	Automatic Dependent Surveillance
ALFUS	Autonomy Levels for Unmanned Systems
ANSP	Air Navigation Service Provider
API	Application Programming Interface
ATC	Air Traffic Control
ATM	Air Traffic Management
CF	Course to a Fix
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off-The-Shelf
DEM	Digital Elevation Model
DF	Direct to a Fix
DME	Distance Measuring Equipment
FA	Fix to an Altitude
FAA	Federal Aviation Administration
FCS	Flight Control System
FPM	Flight Plan Manager
FSM	Finite State Machine
GIS	Geographic Information System
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
HA	Hold/Racetrack to an Altitude
HF	Hold/Racetrack to a Fix
HM	Hold/Racetrack to a Manual Termination
ICARUS	Intelligent Communications and Avionics for Robust UAS
ICAS	International Civil Aviation Organization
IF	Initial Fix
ILS	Instrument Landing System
IMU	Inertial Measurement Unit
MAREA	Middleware Architecture for Remote Embedded Applications
MMa	Mission Manager
RF	Radius to a Fix
RNAV	Area Navigation

RNP	Required Navigation Performance
SCXML	State Chart XML
SID	Standard Instrument Departures
SOA	Service Oriented Architecture
STAR	Standard Instrument Arrival
TCAS	Traffic Collision Avoidance System
TF	Track to a Fix
UAS	Unmanned Aircraft System
UML	Unified Modeling Language
USAL	UAS System Abstraction Layer
VA	Heading to an Altitude
VAS	Virtual Autopilot System
VHF	Very High Frequency
VM	Heading to a Manual Termination
VOR	VHF Omni-directional Range
W3C	World Wide Web Consortium
XML	Extensible Markup Language
XSD	XML Schema Definition Language

1 Introduction

This chapter motivates the need for flight plan and mission specification mechanisms for designing UAS operations, and the means to enable their execution. It also outlines the main contributions of this thesis and presents an overview of the material contained in the following chapters.

1.1 Definition of Unmanned Aircraft System

An Unmanned Aircraft System (UAS) is a system that has, as its central component, an aircraft with no human pilot on-board. Since other components are also required for the unmanned aircraft to be able to operate in a given mission scenario, the term UAS collectively refers to the aircraft and all the other elements supporting its operation.

Typically, an UAS is composed of the following elements:

- Airframe: Or more generally, the mechanical component consisting in an airframe equipped with propeller(s) and servos that operate the control surfaces.
- Flight Control System: A system designed to collect aerodynamic information through a set of sensors (accelerometers, gyros, magnetometers, pressure sensors, GPS, etc.) and to actuate on the propulsion system as well as on the control surfaces in order to automatically direct the aircraft along its flight plan.
- Payload: Formed by the equipment required for the mission. It might include cameras, infrared sensors, synthetic aperture radars, chemical, biological and other types of sensors.

While most UAS will be used as an observation and data gathering platform, some missions may involve acting upon the environment. Crop dusting and cloud seeding are just two examples where UAS actually perform such kind of action.

- Communications Infrastructure: Data links that enable communication between the aircraft and the base station. These data links will provide command and control capabilities, payload data transmission and payload control capabilities between the unmanned aircraft and the ground control station. Finally, they may enable communications between the aircraft and the external world, e.g., other airspace users.
- Ground Control Station: A computer system on the ground designed to monitor and control UAS operation. This system will include flight and payload monitoring, control consoles and decision support tools.
- Launch and recovery system: If special means are required for the aircraft to be launched and recovered these will also be considered part of the system.

UAS can be classified into several categories according to parameters such as weight, altitude, range, endurance, etc. To give an idea of the wide range of existing systems Figure 1.1 partially reproduces an UAS classification from (UVS-International, 2009).

Table 1.1: *UAS categories.*

		Mass (kg)	Range (km)	Flight Alt. (m)	Endurance (h)
μ	Micro	< 5	< 10	250	1
Mini	Mini	< 25 - 150	< 10	150 - 300	<2
CR	Close Range	25 - 150	10 - 30	3.000	2-4
SR	Short Range	50 - 250	30 - 70	3.000	3-6
MR	Medium Range	150 - 500	70 - 200	5.000	6 - 10
MRE	MR Endurance	500 - 1500	> 500	8.000	10 - 18
LADP	Low Alt. Deep Penetration	250 - 2500	> 250	50 - 9.000	0.5 - 1
LALE	Low Alt. Long Endurance	15 - 25	> 500	3.000	> 24
MALE	Medium Alt. Long Endur.	1000 - 1500	> 500	5/8.000	24 - 48
HALE	High Alt. Long Endur.	2500 - 5000	> 2000	20.000	24 - 48

1.2 UAS Potential for Civil Applications

Currently UAS are mostly being used for military purposes, however a wide range of applications have been identified in the civil domain (NASA, 2006; UAVNET, 2005; RTCA, 2007). In general, UAS are specially well suited for the so-called D-cube (Dull, Dangerous, Dirty) applications. The D-cube terminology, which has its origins in the military but is also applicable to civil missions, is defined in (Ingham, 2008) as follows:

- Dull refers to operations that will be too monotonous or require excessive endurance for human occupants (e.g., orbiting above a city for 24 hours while re-broadcasting radio information).
- Dirty refers to hazardous missions that could pose a health risk to a human crew (e.g., monitoring nuclear radiation).
- Dangerous refers to missions that would result in the loss of human life (e.g., deep strike military missions where there is a high risk of hostile fire).

Previous definitions mostly cover situations where UAS can provide services that cannot be accomplished by manned aircrafts. Another important motivator for the introduction of UAS technology is cost. In many cases, manned aircrafts could be replaced by more lightweight vehicles with less associated costs. (UAVNET, 2005) classifies civil UAS applications into five categories:

- Transport, of either cargo or passengers. While passengers transportation may seem a bit far-fetched, it should be noted that current aircrafts are already able to perform most of its flight in auto mode.
- Scientific & Earth observation. Missions in this category include cloud seeding, geological surveys, weather forecasting, atmospheric research, oceanographic observations, etc.
- Surveillance, including flood watch, maritime patrol, volcano monitoring, forest fire detection, oil slick observation, law enforcement, road traffic monitoring, etc.
- Satellite complement, helping overcome some satellite limitations, such as their very constrained orbits and short times spent over a certain area of interest. In this context UAS can help in different ways: as more flexible and precise data collection systems, operating as navigational aids placed at fixed positions, or offering alternative communications solutions.
- Emergencies, including search and rescue, fire fighting, catastrophe situation assessment and disaster operations management.

As seen from previous examples, UAS can be extremely useful as an observation platform. From an operational perspective UAS observation missions can be classified into three categories of increasing complexity:

- Observation of a Fixed Area: A flight path that covers a certain area of interest that does not change during the mission is given before the flight starts. Crop monitoring or disaster damage assessment could be representative of this type of mission. Non-observation missions such as a communications relay application or a point-to-point transport mission would also fall into this category as long as the flight path is completely defined during pre-flight.
- Target Discovery: The UAS system has processing capabilities and is able to recognize some objects or behaviors from the data provided by embarked sensors. Upon target detection the system may perform a holding pattern over the object or try to analyze it in more detail by performing some kind of predefined maneuver. Pipeline inspection, search & rescue and light cargo drops may fall within this category.
- Target Tracking: In this case the system is not only able to detect some target but also to dynamically adapt its flight so that the target is followed. The target may be an specific object, such as a suspicious vehicle, or something larger, like an active fire or an oil slick. In the first example, the aircraft should be able to follow the moving target. In the fire or oil slick examples the aircraft should be able to adapt its trajectory according to a varying perimeter.

In general, the most complex missions include the simplest missions as part of its objective. For instance, in order to track a moving object, its previous discovery is needed, which is based on the inspection of an area. While the previous list may not be complete we believe that it captures the essence of most UAS missions.

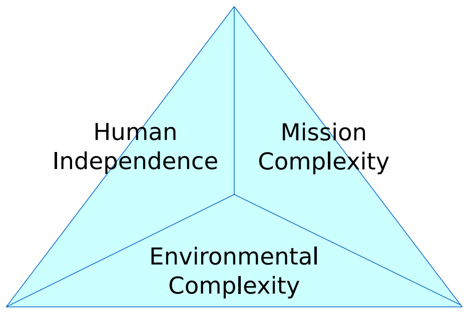


Figure 1.1: *Three axes of the ALFUS Framework.*

1.3 Motivation for Increased Autonomy

An unmanned aircraft can be remote controlled or fly autonomously based on pre-programmed flight plans or even more complex dynamic automation systems.

According to the amount of input required from the operator, UAVs can be classified as ground-controlled, semi-autonomous or fully autonomous (Nas, 2007; Lazarski, 2002). Ground-controlled UAVs require constant input from the operator. However, the trend in unmanned aviation has been towards more autonomous systems and there are very few modern UAVs that are purely remotely piloted. The use of guidance systems is now commonplace. A semi-autonomous UAV can be defined as one requiring ground input only during critical portions of the flight such as take-off, landing and some mission operations. Finally, an autonomous UAV is one where the on-board computer is in control and the operator's task consists mainly in monitoring its systems.

In 2003, a number of unmanned systems professionals from US government agencies and their supporting contractors joined forces to work on the definitions and metrics for unmanned systems autonomy levels. A working group was formed and the ALFUS Framework (Huang *et al.* , 2007) came out as an attempt to come up with a formal framework for a more accurate categorization of autonomy levels of unmanned systems. The ALFUS Framework identifies three axes to consider (see Figure 1.1):

- Mission Complexity (MC): Mission time constraints, level of collaboration required, concurrence and synchronization of events and behaviors, resource management, knowledge requirements, sensory and processing requirements, etc. are aspects to consider when quantifying mission complexity.
- Environmental Complexity (EC): Positioning aids, GPS, markers and other elements can facilitate navigation. Changes in the surroundings, fauna and flora, meteorology, light, terrain and engineered structures, among others, also have an impact on the complexity of the environment.
- Human Independence (HI): The more an unmanned system is able to sense, perceive, analyze, communicate, plan, make decisions and act, the more independent it is.

Autonomy Level actually refers to the Human Independence axis. The other two axes provide context on the type of missions and the environments within which the missions are performed.

The ALFUS framework has laid out two layers of abstraction: the detailed model and the summary/executive model. The executive model, displayed in Figure 1.2, uses a scale that ranges from 0 to 10. At the lowest level the system is able to deal with simple missions in simple

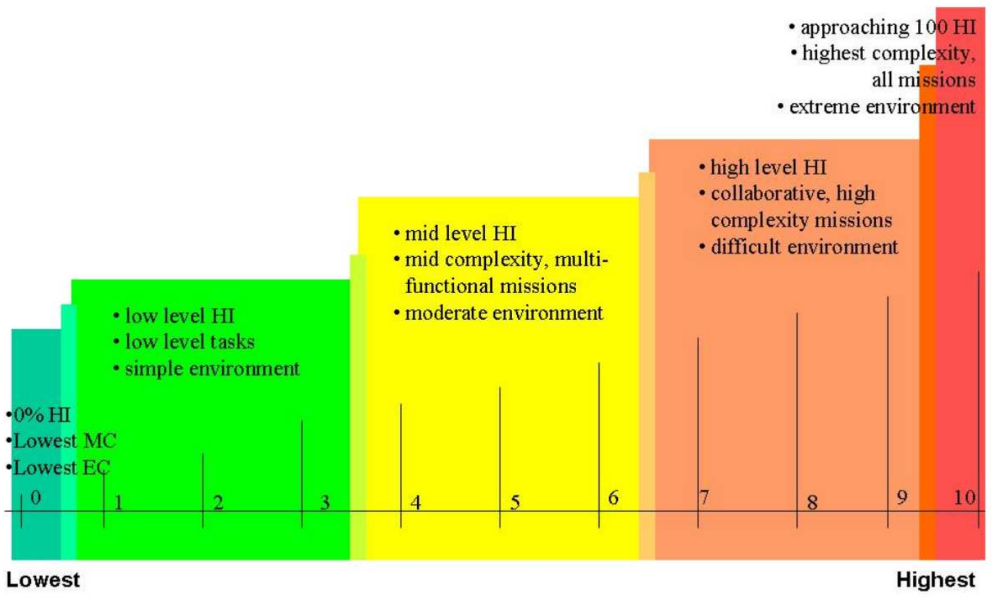


Figure 1.2: *ALFUS Executive Model.*

environments and requires continuous human intervention. At the highest level the system is able to execute complex missions involving coordination of teams in a collaborative manner, with fully real-time planning capabilities in complex dynamic environments and human interaction approaching zero.

There are important motivators for high autonomy in unmanned systems (Huang *et al.* , 2003; Nas, 2008):

- Bandwidth is a limited resource in most circumstances and there may be situations where communications are not available.
- Human operators have poor long-term attention spans and some UAS undertake long endurance missions, which increases the risk of pilot error.
- On-board systems may be able to provide more effective reactions due to its faster access to sensor data and the absence of data-link delays.
- Cost benefits derived from reducing the amount of work of the UAS operators and their training requirements.

The aim of the work presented in this thesis is to provide a mid to high level of autonomy to civil UAS restricting ourselves to single vehicle systems. To this end, flight plan and mission management layers are added on top of a commercial-off-the-shelf flight control system. These management components form part of a wider hardware/software architecture that enables the UAS to be reconfigured for its adaptation to different mission scenarios. We believe this flexibility to be a crucial requirement for UAS to be successful in the civil domain. Hence, we can restate our goal as trying to achieve a high level of autonomy while keeping reconfigurability of the system equally high.

As previously seen, the level of autonomy is deeply interrelated with environment and mission complexity. When it comes to the use of UAS for civil missions some aspects are considerably simplified. The environment axis becomes less of an issue as long as the system is able to fly at a safe altitude. General availability of satellite navigation systems can also be assumed. A critical issue, and perhaps the most important setback to proliferation of civil UAS from a technical perspective, is the lack of an effective and affordable collision avoidance system (DeGarmo, 2004). This issue is not addressed in this thesis, where we assume sense-and-avoid capabilities to be already available. It can also be considered a less stringent requirement if the UAS operates in segregated airspace or with a ground operator in permanent contact with air traffic control authorities.

With regard to the kind of missions to be performed, most times being able to scan an area or follow a given path for taking a number of measures will suffice. It is possible that the area of interest experiments variations during the mission. A more complex mission may involve searching and perhaps following a moving target. Although we can certainly imagine much more complex scenarios, we believe that a system able to perform efficiently in these general situations can provide great value.

1.4 Thesis Contributions

A flight plan describes the path that is going to be followed by an aircraft. In the case of UAS, when not being remotely piloted, most systems rely on a list of waypoints for the specification of their flight plan. Each waypoint corresponds to a geographical position defined in terms of latitude/longitude coordinates. As it will be seen, this approach has several important limitations and we believe that it can be greatly improved.

Moreover, there is a lack of mechanisms for mission specification and execution that renders current systems only suitable for the specific application they where designed for.

The main contributions of this thesis are:

- A new concept for specifying UAS flight operations that borrows the leg and path terminator approach used in Area Navigation (RNAV) (FAA, 2008; EUROCONTROL, 2003) and extends them for a better adaptation to UAS requirements. Extensions include the addition of control structures that enable repetitive and conditional behavior, and also parametric legs, that can be used to generate complex paths from a reduced number of parameters. The proposed flight plan specification concept gets materialized in the definition of an XML (Bray *et al.* , 2006) based language.
- The definition of update mechanisms to modify the flight plan during flight and dynamically adapt the trajectory to the mission needs.
- The design and implementation of the Flight Plan Manager (FPM), an embarked software component that manages execution of the flight plan. The FPM provides a wide set of operations that can be used by both human operators and other UAS components to control the UAS flight. To take advantage of current off-the-shelf Flight Control Systems the structures included in the flight plan are translated to waypoint navigation commands. In this way, the advanced capabilities provided by the flight plan specification language get implemented as a new layer on top of existing technologies.
- An architectural model that promotes separation of flight and mission concerns. This separation allows a single flight plan to be reused across missions. Not enforcing the presence of a mission control component, it also permits an incremental approach to designing and operating an UAS.
- A reconfigurable mission management system with State Chart XML (SCXML) (W3C, 2009) as the proposed language for specifying UAS behavior related to the mission. SCXML is a working draft developed by the World Wide Web Consortium (W3C) and is based on Harel's State Charts (Harel & Politi, 1998), a widely used language for modeling complex behavior.

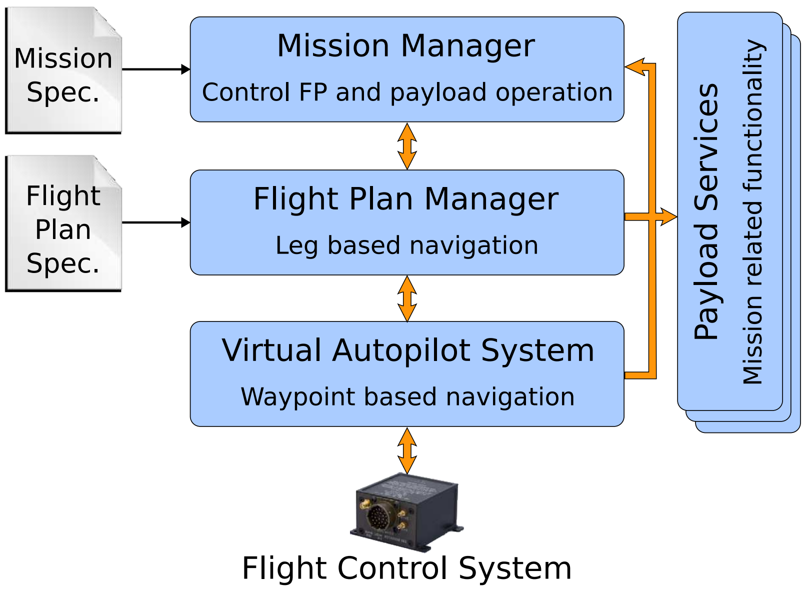


Figure 1.3: *Relationship between MMa, FPM, VAS and other payload.*

All proposed mechanisms for both specification and execution are integrated into the service oriented architecture described in Chapter 3. Figure 1.3 shows the two services developed as a result of this thesis work, namely the Mission Manager (MMa) and the Flight Plan Manager (FPM). The figure also shows their interactions with the Virtual Autopilot System (VAS) and Payload Services. The VAS provides an standardized interface for accessing the Flight Control System. Payload Services represent all the different services involved in the execution of the mission.

As seen in Figure 1.3, the FPM receives a document that contains the flight instructions in our proposed flight plan specification language. It then translates the flight plan description into an internal representation and performs its execution by sending navigation commands to the VAS. The MMa interacts with the FPM to determine which part of the flight plan is under execution and coordinates payload operation accordingly. It also takes into account the data provided by payload services and makes use of the update and control mechanisms provided by the FPM in order to adapt the flight plan to the mission needs. The MMa operates in an event-driven fashion making progress through the states of the automaton defined in an SCXML document.

1.5 Thesis Organization

Chapter 2 presents the state of the art regarding flight plan and mission specification mechanisms. Then, in Chapter 3, a description of the system architecture that accommodates the flight plan and mission components can be found.

Chapters 4 to 6 cover the flight plan related part of this thesis. In Chapter 4, the proposed language for specifying UAS flight plans is presented. Chapter 4 also includes the definition of the mechanisms to dynamically adapt the flight to the mission needs. Chapter 5 describes the design and implementation of the Flight Plan Manager service, which is the module responsible for carrying out the execution of the flight plan. The results related to the flight plan definition and its execution are shown in Chapter 6.

Chapters 7 and 8 cover the mission related part. Several mission specification methods already exist, most of them tied to a given platform. In this thesis we propose using State Chart XML for this purpose. An overview of the language and the implementation of a proof-of-concept prototype of the Mission Manager can be found in Chapter 7. The Mission Manager is the module that coordinates operation of the different UAS components for the achievement of the mission goals. Chapter 8 shows the results of the combined operation of the flight plan and mission managers.

Finally, Chapter 9 presents conclusions and future research.

A number of appendices provide additional information with regard to the flight plan specification language and mission examples used throughout the dissertation. Appendix A contains the definition of the XML schemas that determine the structure of flight plans and their updates. These schemas can be used to validate syntactic correctness of flight plan and update specifications. Appendix B contains the complete flight plan of the flight inspection mission used in Chapter 6. Appendix C provides both the flight plan and SCXML specifications of the example mission used in Chapter 8.

2 Previous Work

This chapter covers the state of the art in the two areas that this thesis is concerned about: (1) specification and execution of the flight plan, and (2) specification and execution of the UAS mission, where non flight related payload operation is also taken into account.

The problem of flight plan specification for UAS systems is addressed from two perspectives. First, in Section 2.1, the capabilities of current commercial UAS autopilots are discussed. Afterwards, in Section 2.2, we have a look at practices in commercial aviation for specifying flight plans in a way that suits computerized systems.

But the flight plan alone does not suffice for specifying a complete mission, where sensed inputs and payload operation need to be considered in order to achieve the mission goals. In Section 2.3, some fundamental concepts about autonomous mobile robot architectures are introduced. After that, Section 2.4 presents several mission specification techniques used in Unmanned Aircraft Systems and the broader field of Unmanned Autonomous Vehicles.

2.1 COTS Autopilot Capabilities

The main purpose of an UAS autopilot system is to control the aircraft flight with minimal human intervention. A powerful autopilot may be able to execute all phases of a mission fully autonomously. A less capable system may require manual take-off and landing. In the worst case, it may only provide aircraft stabilization requiring continuous remote control.

In their survey of autopilots for small fixed-wing UAS (Chao *et al.* , 2007) H.Chao et al. describe the main features of a typical off-the-shelf autopilot. In most cases, the system comprises a GPS receiver, a micro inertial guidance system and an onboard processor (state estimator and

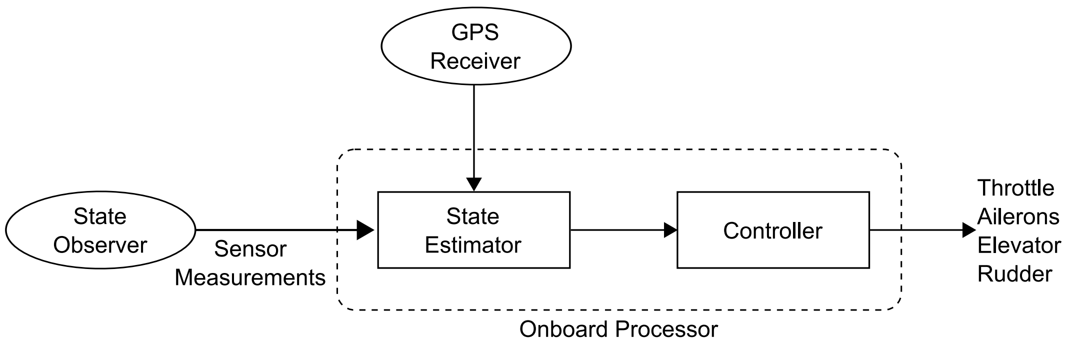


Figure 2.1: *Functional Structure of an UAS Autopilot System.*

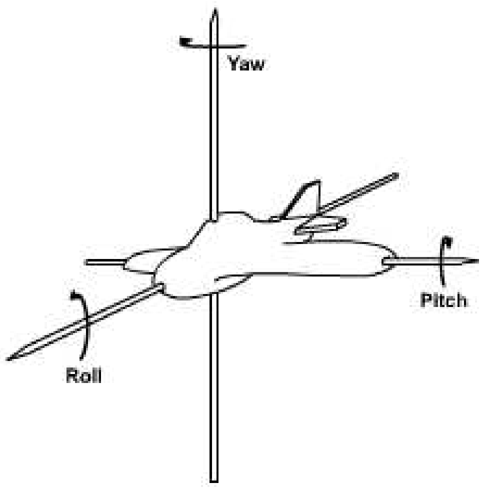


Figure 2.2: *Roll, pitch and yaw rotations about the respective axes.*

flight controller) as illustrated in Figure 2.1. The inertial guidance system together with the GPS receiver provide a complete set of sensor readings like absolute aircraft position, aircraft attitude, accelerations, pressures, etc. The onboard processor uses this data to estimate the aircraft state and operate on the actuators that control the vehicle behavior.

Most current commercial and research autopilots focus on GPS based waypoints navigation. The path-following control of the UAS can be separated into different layers:

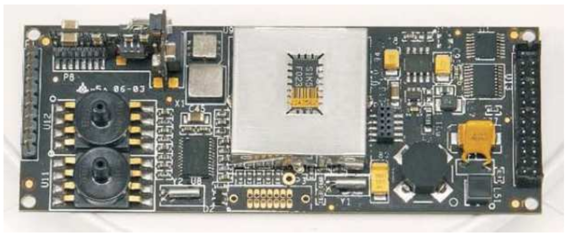
1. An inner loop on roll and pitch for attitude. These are the rotation angles about the aircraft's longitudinal and lateral axes respectively.

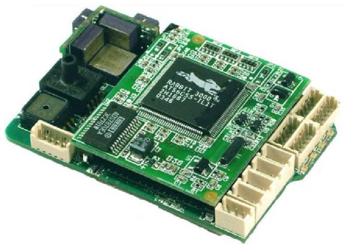
2. An outer loop on yaw and altitude for trajectory or waypoints. Yaw, which is also know as heading, refers to the angle of rotation about the aircraft's vertical axis. As shown in Figure 2.2, the combination of roll, pitch and yaw determines the vehicle's orientation.

3. Finally, one last loop controls waypoint navigation.

There is an increasing amount of autopilot manufacturers providing solutions for UAS. Figure 2.3 displays some of the available systems, namely Procerus Technologies Kestrel Autopilot (Procerus Technologies, 2009), MicroPilot MP2028 Series (MicroPilot, 2009), Cloud Cap Technology Piccolo Systems (Cloud Cap Technology, 2009) and UAV Navigation AP04 (UAV Navigation, 2009), but many others can easily be found. Virtually all of them, even those

(a) *UAVNavigation AP04*


(b) *MicroPilot MP2128g*


(c) *Procerus Technologies Kestrel*

(d) *Cloud Cap Piccolo Plus*

Figure 2.3: *Autopilot systems from different vendors.*

targeting a hobbyist audience such as Attopilot (EM Technologies Group, 2009) or Ardupilot (Anderson, 2009), support waypoint based navigation. To control the aircraft's trajectory, the user specifies a list of waypoints defined in terms of their latitude/longitude coordinates and the aircraft flies them in sequence.

These systems may differ in many aspects: performance, reliability, level of integration with other payload, number of inputs and outputs available, and features of their corresponding ground control stations (if there is one), among others. Nevertheless, they tend to offer similar capabilities from a functional point of view, and waypoint navigation is definitely a common denominator to the vast majority of them. Differences at this level can be found mainly in their ability for automatic take-off and landing and its available modes of operation apart from waypoint navigation and full remote control.

While providing a list of waypoints may suffice for simple observation missions, UAS have potential for being used in more complex scenarios and we believe that providing a list of waypoints is not the best way for describing their trajectories. An interesting example that goes beyond that is the Paparazzi Project (Brisset *et al.* , 2006; Paparazzi, 2010), that offers its own flight plan specification language.

Paparazzi's flight plan specification language has a rich set of primitives for commanding the aircraft to operate in different navigation modes. These modes are used to keep a fixed attitude, to keep a given course, to go to a given waypoint or to circle around one. It supports *goto* directives and constructs that enable looping and also permits to define navigation procedures in the C (Kernighan & Ritchie, 1978) programming language or as a combination of basic primitives. All this elements are organized in units, called blocks, that represent each part of a mission. A small example of a Paparazzi's flight plan is shown in Listing 2.1.

Listing 2.1: Example of a Paparazzi flight plan.

```
<flight_plan alt="75" ground_alt="0" lat0="43.46223" lon0="1.27289"
   max_dist_from_home="1500" name="turing complete">
   <waypoints>
      <waypoint name="HOME" x="0" y="0"/>
      <waypoint name="STDBY" x="9.4" y="162.3"/>
      <waypoint name="2" x="23.7" y="123.1"/>
   </waypoints>

   <exceptions>
      <exception cond="estimator_z > 300" deroute="wait"/>
   </exceptions>

   <blocks>
      <block name="start">
         <go wp="STDBY"/>
      </block>
      <block>
         <for from="1" to="5" var="i">
            <set value="\$i*750*cos(RadOfDeg(30)" var="waypoints[WP_2].x"/>
            <set value="\$i*750*sin(RadOfDeg(30)" var="waypoints[WP_2].y"/>
            <go hmode="route" wp="2"/>
            <set value="\$i*750*sin(RadOfDeg(30))" var="waypoints[WP_2].y"/>
            <circle radius="nav_radius" until="NavCircleCount()>1" wp="2"/>
         </for>
      </block>
      <block name="wait">
         <circle radius="nav_radius" wp="STBY"/>
      </block>
   </blocks>
</flight_plan>
```

The approach followed by the Paparazzi Project has many similarities with ours, but there are some significant differences too. Paparazzi's specification language tries to take full advantage of the capabilities of a single autopilot, whereas our intend is to provide a mechanism that can operate on a wide range of autopilot systems. Another difference relies in the source of inspiration for the specification language. Both of them being based on XML (Bray *et al.* , 2006), many elements of the Paparazzi's language resemble constructions that can be found in a general purpose programming language as C. In fact, all the contents of the flight plan are compiled into a program that is later on executed by the autopilot. In our case, we gravitate towards specification primitives used in commercial aviation, which eventually could facilitate integration in non-segregated airspace. Finally, having the flight plans compiled into binary code limits the system's ability for supporting in flight updates. Both projects also differ in the types of systems they target, while the Paparazzi autopilot is specially well suited for small vehicles (small enough to be man-portable), our target are bigger systems with less restrictions on computing resources.

In this section, we have seen the most common navigation capabilities of current off-the-shelf autopilot systems and identified waypoint based navigation as being generally available. It is our goal to provide semantically richer and more flexible specification primitives for flight plan specification and the appropriate means for their execution. In the following chapters, we will discuss our proposed specification language and its execution mechanism. Reliance on the Virtual Autopilot System (see Chapter 3) will enable operation on top of existing off-the-shelf systems. Before that, next section provides an overview of RNAV, an advanced navigation method being used in commercial aviation.

2.2 A Look at Commercial Aviation

Previous section described the state of the art with regard to what current commercial-off-the-shelf autopilots allow for. The purpose of this section is to explore current practices in commercial

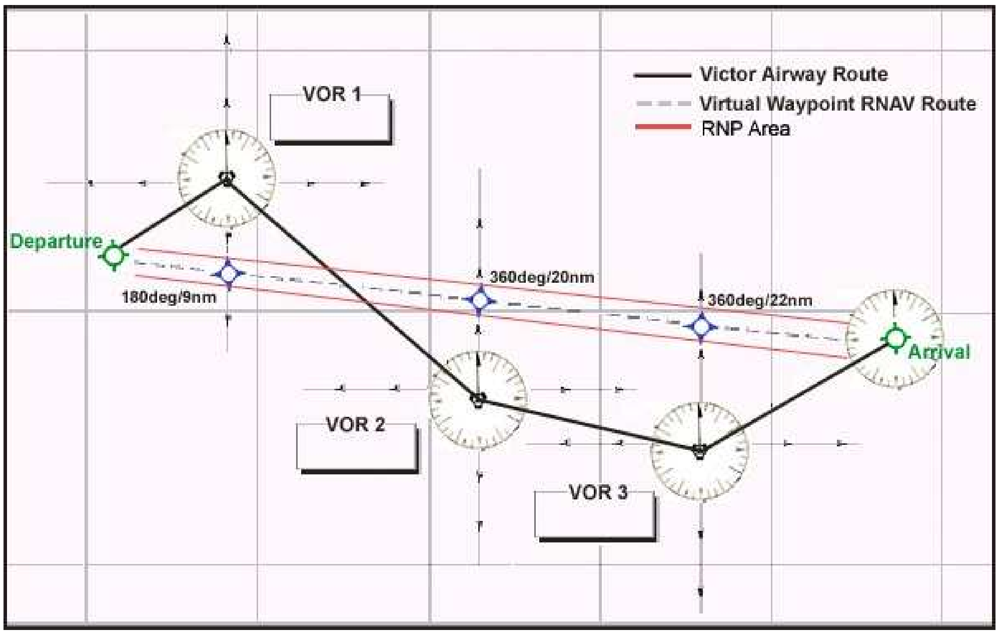


Figure 2.4: *RNAV versus non-RNAV navigation.*

aviation and see what is going on in terms of aircraft navigation in this domain. In particular, we focus on Area Navigation (RNAV), a method of navigation that allows aircraft operation on any desired flight path.

Navigation aids (navaids) are systems that use station-referenced radio frequencies to permit determining the position of an aircraft. With traditional methods navigation is restricted to direct trajectories to or from specific ground-based navaids. Area navigation (RNAV) (Airbus, 2002) takes advantage of the increasing amount of navaids, such as VHF Omni-directional Range (VOR), Distance Measuring Equipment (DME) and Global Navigation Satellite System (GNSS), among others, to automatically determine the aircraft position. With RNAV, virtual waypoints that are not tied to specific navaids can be defined. Doing so enables the aircraft to follow any desired flight path.

Figure 2.4 compares an RNAV procedure, displayed as a dashed line, against its non-RNAV counterpart, displayed as a solid line connecting VOR stations. In the RNAV procedure, data obtained from the different available navaids is combined in order to specify several virtual waypoints along a direct track from the departure position to the destination. This trajectory is clearly more optimal than the VOR to VOR alternative.

RNAV enables better use of the available airspace, providing benefits such as flight time reduction, less fuel consumption, congestion reduction (less flight delays) and fewer acoustic pollution. Although these advantages can also apply to UAS, our interest in RNAV comes from the fact that it already provides well established means for specifying flight trajectories that we can learn from.

2.2.1 Specification of RNAV Procedures

In commercial aviation an aircraft goes through several phases during a flight, performing a set of procedures at each phase. Terminal area procedures, Standard Instrument Departures (SID), Standard Instrument Arrivals (STAR) and Approach procedures have traditionally been described in Aeronautical Information Publications using charts and associated text (see Figure 2.6). However, an aircraft navigation system must be provided with route data in a format that can be processed by a computer. ARINC 424 Navigation System Data Base Standard (Aeronautical Radio, Inc., 2008) is an international standard used to supply computerized navigation systems, flight planning systems and simulators with navigation data.

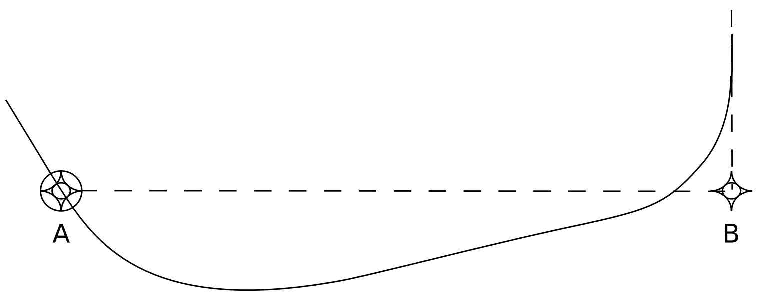
In order to achieve the translation of the text and the routes depicted on charts into a code suitable for navigation systems, the industry has developed the ”Path and Termination” concept (EUROCONTROL, 2003). Each flight phase (departure, arrival, approach ...) is divided into smaller chunks called legs, and each leg describes the desired path to reach a termination point. 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.). Path Terminator codes should be used to define each leg of an RNAV route from take off until the en-route structure is joined and from the point where the aircraft leaves the en-route airway until the end of the planned flight.

Nowadays, there are 23 different Path Terminator codes, although most navigation systems only implement a sub-set of these. Besides, not all of them are acceptable for RNAV use (EUROCONTROL, 2003). The leg types that can be used in RNAV procedures are listed below:

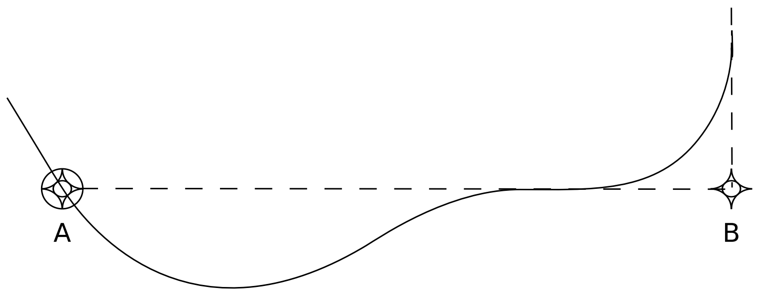
- IF - The Initial Fix defines a database fix as a point in space. It is only required to define the beginning of a route or procedure.
- CF - Course to a Fix defines a specified course to a specific database fix. Course meaning the intended direction of flight in the horizontal plane measured in degrees from north (FAA, 2009).
- DF - Direct to a Fix defines an unspecified track starting from an undefined position to a specified fix.
- FA - Fix to an Altitude defines a specified track over ground from a database fix to a specified altitude at an unspecified position.
- VA - Heading to an Altitude termination defines a specified heading to a specific Altitude termination at an unspecified position.
- VM - Heading to a Manual Termination defines a specified heading until a Manual termination.
- TF - Track to a Fix Leg defines a great circle track over ground between two known databases fixes.
- RF - Constant Radius Arc Leg defines a constant radius turn between two database fixes, lines tangent to the arc and a center fix
- HF, HM, HA - Hold/Racetrack to a Fix define racetrack pattern or course reversals at a specified database fix. HF is used for single circuit terminating at the fix (base turn). HM and HA are respectively Manual and Altitude terminated.

Some of the previous leg types, the ones that have been deemed appropriate for UAS applications, are further discussed in Chapter 4.

To give a flavor of the behavior of different leg types Figure 2.5 shows an example where the path followed with a DF leg is compared to the one followed with a TF leg. Figure 2.5a illustrates a situation where a DF leg is used to connect two consecutive waypoints, resulting in the aircraft flying directly from A to B. In Figure 2.5b a TF leg is used instead and, in this case, the aircraft intercepts the track that connects the two waypoints before reaching B. There is a difference between the two waypoints that should also be noted. A is surrounded by a



(a) *Direct to a Fix*



(b) *Track to a Fix*

Figure 2.5: *Comparison of two different leg types.*

circumference indicating that it is a fly-over waypoint and, therefore, the aircraft must fly over it before initiating the turn. This contrasts with the turning maneuver at *B*, that starts earlier because *B* is a fly-by waypoint.

RNAV provides concepts that enable rich specification of flight plans, but still needs some adaptation for supporting UAS missions. Following chapters will present our proposed flight plan specification language and the execution mechanisms that enable its operation on top of currently available autopilot systems.

2.2.2 Flight Management System

The Flight Management System (FMS) holds the flight plan and controls aircraft navigation, it also allows the pilot to modify the flight plan as required in flight. The FMS is usually composed of two parts: a computer unit and a Control Display Unit (CDU). The computer unit provides the computing platform and various interfaces to other avionics. The CDU, which usually consists in a small screen accompanied by a keyboard, provides the primary human/machine interface for data entry and information display (see Figure 2.6).

The capabilities of flight management systems can vary due to the differences in target markets. These capabilities range from simple point-to-point lateral navigators to highly sophisticated functions such as four-dimensional trajectory prediction and performance computations. An overview of the main functions that can be provided by a FMS follows (Spitzer, 2007):

- Flight planning: The flight plan specifies the route the aircraft will follow. It is generally determined on the ground, before departure, and constructed by linking data stored in the navigation database. This data may include departure and arrival procedures, airways, prestored company routes, fixes and crew-defined fixes. The crew can modify the flight plan at any time, changes can also be submitted from on-ground offices via a datalink. During preflight, other relevant information such as gross weight, fuel weight, and weather forecast

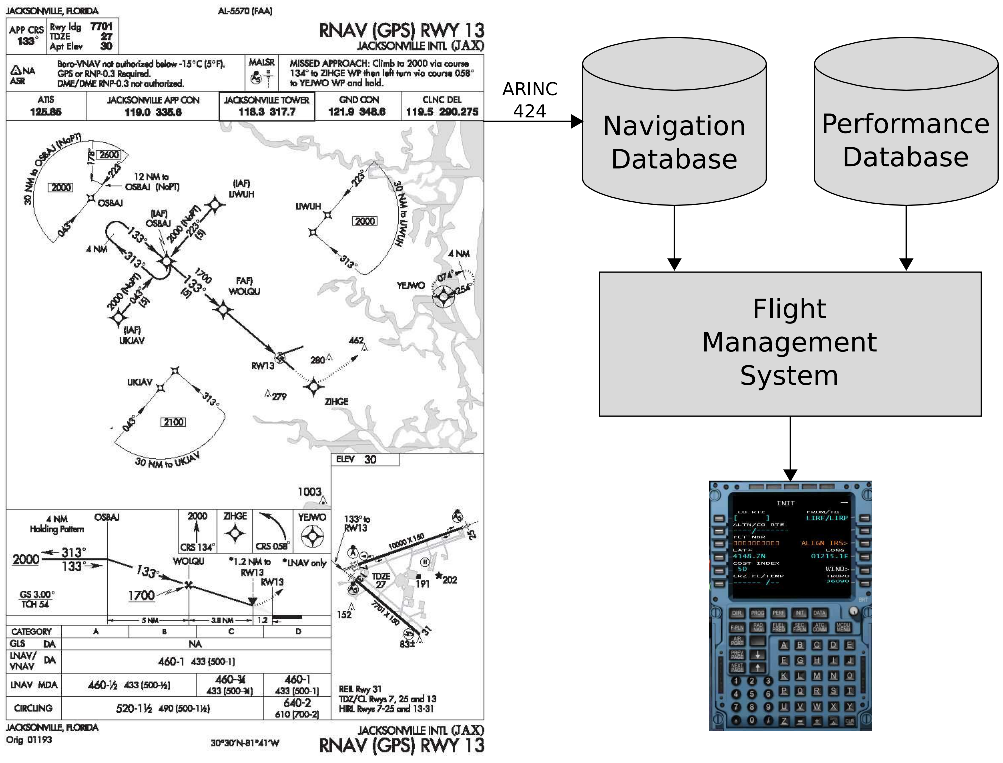


Figure 2.6: *Flight Management System.*

is also provided.

- Navigation: Navigation is about the system's ability to determine the current state of the aircraft. This state usually consists in its three-dimensional position (generally WGS-84 geodetic coordinates), velocity vector, altitude rate, track angle, heading and drift angle, wind vector, estimated position uncertainty and time. This data is obtained by combining data from both navigation receivers and autonomous sensors. The accuracy of the estimated position determines the aircraft's navigation performance. A minimum Required Navigation Performance (RNP) may be needed to operate within a defined airspace.

- Guidance: Given the flight plan and the aircraft's position, the FMS calculates the course to follow. The guidance function is responsible for producing commands to guide the aircraft along both the lateral and vertical profiles. The FMS typically computes roll axis, pitch axis, and thrust axis commands to guide the aircraft.

- Trajectory prediction: This function is responsible for computing the predicted four-dimensional flight profile of the aircraft. Lateral path, fuel, time, distance, altitude and speed are obtained for each point in the flight plan.

- Performance: It provides the crew with all sorts of information to help optimize the flight and access to aircraft's specific performance data. A few examples include optimal speed computations for minimizing time and fuel consumption, maximum and optimum altitudes, thrust limits, take off speeds, etc.

To accomplish its functions the FMS interfaces with a wide array of other avionic systems, such as navigation sensors, displays, the flight control system, the engine and fuel system and the data link system.

2.3 Mobile Robot Control Architectures

Three elements are identified as necessary to fully specify a mission (Ulam *et al.* , 2006):

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 in later chapters, consists in monitoring a burned area in search of hotspots. This mission is composed of separate tasks, such as patrolling the 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 proceeding to its analysis.

Traditionally, there have been two approaches to mobile robot control (Arkin, 1998):

Deliberative Control: The main characteristic of deliberative systems is that they rely on a representation of the world which serves as the basis for predicting and making decisions about subsequent actions.

Reactive Control: Pure reactive systems lack this representation and their actions arise as a direct response to stimuli.

The main drawbacks of deliberative methods are their lower response times and that the internal world representation may rapidly become obsolete. However, a purely reactive system may not be capable of dealing with complex tasks. It is common practice to design autonomous mobile robots as hybrid systems combining low-level reactive behaviors with higher level deliberation and reasoning. Reactive control is located closest to the system actuators, and is given highest priority. Nevertheless, deliberative control is given precedence when the reactive component cannot handle a certain situation the mobile robot is confronted with.

As depicted in figure 2.7, hybrid systems are usually modeled as having three layers: one deliberative, one reactive and one middle layer (Orebäck, 2004).

The Reactive Layer of a hybrid system often consists of separate behaviors running in parallel, where each behavior has one specified task. Example behaviors are goto-goal and avoid-obstacles. Reactive behaviors represent a tight coupling from the sensors to the actuators. In some architectures reactive behaviors are hierarchically organized with more complex behaviors being obtained by a combination of simpler ones.

The Deliberative Layer handles mission planning and reasoning, localization, path planning and interaction with human operators. Tasks in this layer are allowed to be computationally expensive and therefore take relatively long time. The skills and complexity that are needed in the deliberative layer are highly related to the amount of autonomy one is seeking.

The middle layer, often called the Sequencer Layer, or supervisory layer, bridges the gap between the deliberative and reactive layers. Its basic function is to rewire the reactive layer according to a global state obtained from the deliberative layer, thus deciding which is the set of behaviors that should be running. It should monitor the reactive layer and be informed as progress is made.

Narayan et al. (Narayan *et al.* , 2007) provide an interesting survey of different robotics and UAS architectures confirming that a vast majority of them implement a hybrid architecture consisting in some variation of the presented model.

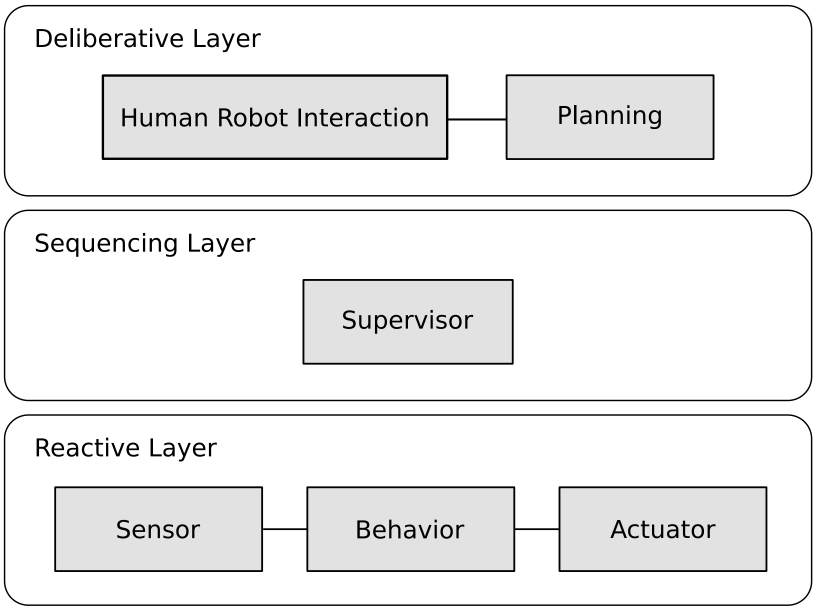


Figure 2.7: *Hybrid system layers*

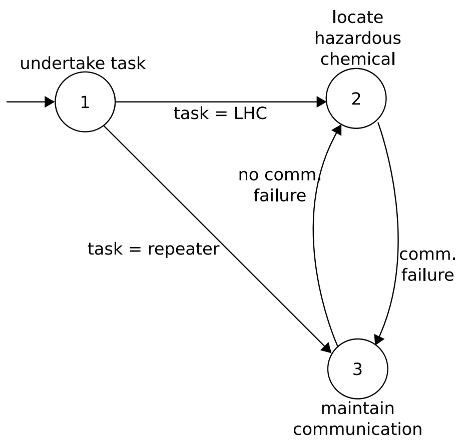
As described in Chapter 3, our UAS implements a highly distributed architecture. Nevertheless, we can still identify components providing basic reactive behaviors, such as the VAS, the FPM and payload related services, and higher level deliberative services such as a Long Term Planner. The MMa sits in between enabling definition of higher level reactive behaviors and managing its execution.

Although deliberative services will eventually be added to the system, in this thesis we focus on providing a highly reconfigurable purely reactive control layer.

2.4 UAS Mission Control

This section presents the state of the art related to mission specification of Unmanned Aircraft Systems and Unmanned Autonomous Vehicles in general. The selected references are representative of different approaches for specifying the behavior of a reconfigurable systems.

Apex is a NASA Open Source Software architecture and development toolkit for creating intelligent, autonomous agents. Apex (NASA, 2009) is a variant of the three-tier type architecture. It has been used in diverse applications, including the two UAS efforts described in (Freed *et al.* , 2005). Constructing a new autonomy application with Appex involves two main steps. The first one is integration with the controlled architecture. This step involves enabling communications with the controlled system by means of the required protocols. Also as part of this initial step, Apex primitives are defined. These primitives represent command outputs from Apex to the controlled system. The second step in constructing an Apex application is to specify desired autonomous behavior. Apex operates in a goal-driven fashion, where procedures are used to specify how a given goal should be achieved. Both primitives and procedures are described using Apex's Procedure Description Language. In the example shown in Listing 2.2, a procedure is defined to image a ground target using a high-resolution fixed-angle camera called camera-1. Procedure subtasks, specified using the step clause, are not necessarily carried out in listed order. Instead, they are assumed to be concurrently executable unless otherwise specified.



(a) *Plan expressed as FSM.*

```
oblig on task(LHC) do
  /smc/tasks.init(LHC)

oblig on task(Repeater) do
  /smc/tasks.init(Maintain_communication)

oblig on communication_failure do
  /smc/tasks.switchto(Maintain_communication)

oblig on no_communication_failure do
  /smc/tasks.switchto(LHC)
```

(b) *Same plan in terms of policies.*

Figure 2.8: *Comparison between FSM and policy representations.*

Listing 2.2: Definition of an Apex procedure.

```
(procedure
  (index (get hires image ?target))
  (profile camera-1)
  (step s1 (move-to best-imaging-loc for ?target => ?loc))
  (step s2 (power-up camera-1))
  (step s3 (orient-camera-to ?target)
      (waitfor (:and (ready camera-1)(location +self+ = ?loc))))
  (step s4 (take-picture camera-1)
     (waitfor (end ?s3)))
  (restart-when (task-state +this-task+ = resumed))
  (end-when (image-in-memory ?target))
```

A different approach is presented in (Asmare *et al.* , 2006). In this case, a mission is specified in terms of roles, tasks to be performed by a role and the policies for managing tasks. Roles would be distributed among different components of an autonomous vehicles team, however the same principles apply for a single autonomous vehicle. Policies are used to specify tasks to be carried out by a role as well as privileges regarding access to services provided by other roles or shared resources. Policies (Damianou *et al.* , 2001) are rules governing choices in behavior. This work focuses on two main types of policies: obligation and authorization. Obligations are event-condition-action rules and authorization policies define what actions a subject can perform on a target resource or service. Figure 2.8 shows how obligation policies can be used to encode the vehicle behavior depicted in the accompanying FSM.

M. Barbier et al. opt for Petri Nets as the method of choice for UAS' mission specification. (Barbier & Chanthery, 2004) presents an architecture consisting of (1) a set of Petri nets that hierarchically models the vehicle behavior, (2) three software programs carrying out the decisional tasks and (3) a supervisor managing the update of the vehicle behavior and the communication with the decisional tasks. The software programs carrying out the decisional tasks are a planning program that computes the optimal plan according to the mission and its constraints, a guidance program that calculates the control commands sent towards the vehicle and a third program which centralizes dynamic data management. Figure 2.9 displays a Petri net that models an observation mission. The place marked in this Petri net indicates the phase in which the vehicle is or the high level action in progress. Each phase can be broken up in an increasingly detailed way with piloting controls located at the lowest level of this decomposition. Petri nets detail the vehicle behavior during nominal mode and in the degraded situations.

In previous examples, a mixture of text based languages and graphical representations of

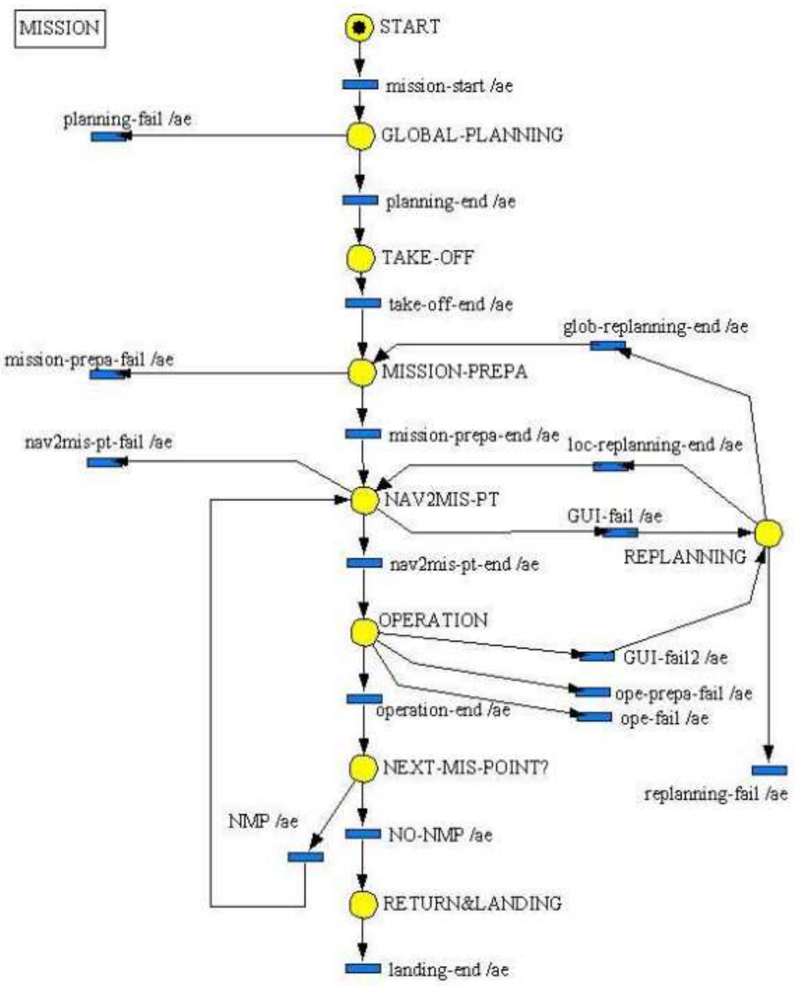


Figure 2.9: *High level mission petri net.*

vehicle behavior have been outlined. A common theme of the following references is their use of FSMs as the means for describing the vehicle behavior.

The WITAS project (Doherty *et al.* , 2004) aims at developing a prototype distributed architecture for autonomous unmanned aerial vehicle experimentation. To this end, it implements a software architecture, known as the Modular Task Architecture (MTA) where deliberative, reactive and control components interact in a distributed and concurrent manner. A task is defined as a behavior intended to achieve a goal in a limited set of circumstances and is implemented by means of a Task Procedure (TP). TPs use CORBA to communicate with one another. In fact, a TP is any CORBA object that implements the Witas::Task interface and adheres to some behavioral restrictions. To facilitate its development and avoid the complexities of CORBA an XML based Task Specification Language (TSL) is used for defining TPs.

A central component of the WITAS vehicle is the Primary Flight Control (PFC) system, which supports several control modes: take-off, hovering, dynamic path following and reactive flight modes for interception and tracking. Task procedures in the Deliberative/Reactive system issue commands to the PFC system and receive aircraft (in this case, an helicopter) states and events from it.

The TSL (Nyblom, 2003) provides some tags to declare required parameters, local variables and CORBA objects the TP interacts with. Other tags are used to specify what actions should be performed at several stages in the life cycle of a TP. The main element for specifying behavior is

```
<tp name = Taskname>
<declarations>
   <parameter .../> ... <parameter .../>
   // other declarations, e.g., local
   // variables and constants
</declarations>
<services>
   // CORBA server objects,
   // event channels used, etc.
</services>
<init>
   // Host code for task
   // specific initialization
</init>
<destroy>
   // Host code for task specific cleanup
   // CORBA cleanup handled automatically
</destroy>
<start>
   // Executed with call to TP start() method
   // Host code plus host code macros
   // Typically will perform some setup then
   // a jump to FSM state
</start>
<fsm>
   // Main behavioral specification in form
   // of a finite state machine
</fsm>
</tp>
```

(a) *TSL tags and partial schematic for a TP specification.*

```
<fsm>
<state name = sname>
   <action>
      // Executed whenever TP
      // enters this state
   </action>
      // State specific reactions
      // to events
   <reaction event = "event name">
      ...
   </reaction>
      ⋮
   <reaction event = "event name">
      ...
   </reaction>
</state>

//More state specifications ...

// Global reactions to events
<reaction> ... </reaction>
   ⋮
<reaction> ... </reaction>
</fsm>
```

(b) *TSL tags and partial schematic for an FSM specification.*

Figure 2.10: *TP Specifications with TSL tags.*

an *fsm* tag that is used to encode a Finite State Machine (FSM) with the user defined states the TP goes through while running. Figure 2.10 illustrates how TSL is used. Blank spaces within elements should be filled with C++ or Java code. The TP specification is translated to C++ or Java and compiled before being deployed to the vehicle.

Preceding the WITAS project, another example where FSMs are used corresponds to (Mackenzie *et al.* , 1997). In this case, an entity capable of stimulus-response behavior is referred to as an agent. The simplest agents correspond to primitive behaviors like sensors, actuators and motors. More complex agents can be recursively constructed via assemblage. An assemblage is a coordinated society of agents which function as a new agent. Coordination may imply temporal sequencing (specified using Finite State Automata), cooperation (concurrency) or competition. These relationships are defined using an architecture independent Configuration Description Language (CDL). CDL is used to specify the instantiation and coordination of primitives. A toolset called MissionLab is provided in order to enable the specification of the mission as a hierarchical combination of reactive components. Figure 2.11 shows MissionLab's Configuration Editor, which is a graphical tool for building a mission with a set of robot behaviors. As in the WITAS case, behavior descriptions are compiled into executable code before its deployment onto the autonomous platform.

Finally, in (Dong & Sun, 2004; Dong *et al.* , 2007) Dong et al. describe the design and implementation of a behavior-based architecture for unmanned aerial vehicles. In this architecture, a behavior is defined as a function between the sensing input and the action output. Behaviors can be combined to define an schema, which consists in a set of arcs and nodes where each node represents a behavior and arcs represent transitions between nodes (see Figure 2.12). Transitions are triggered by events. Schemas can be hierarchically organized. The following behaviors are defined: ready, move, accelerate, decelerate, lift/descend, head to, hover, track,

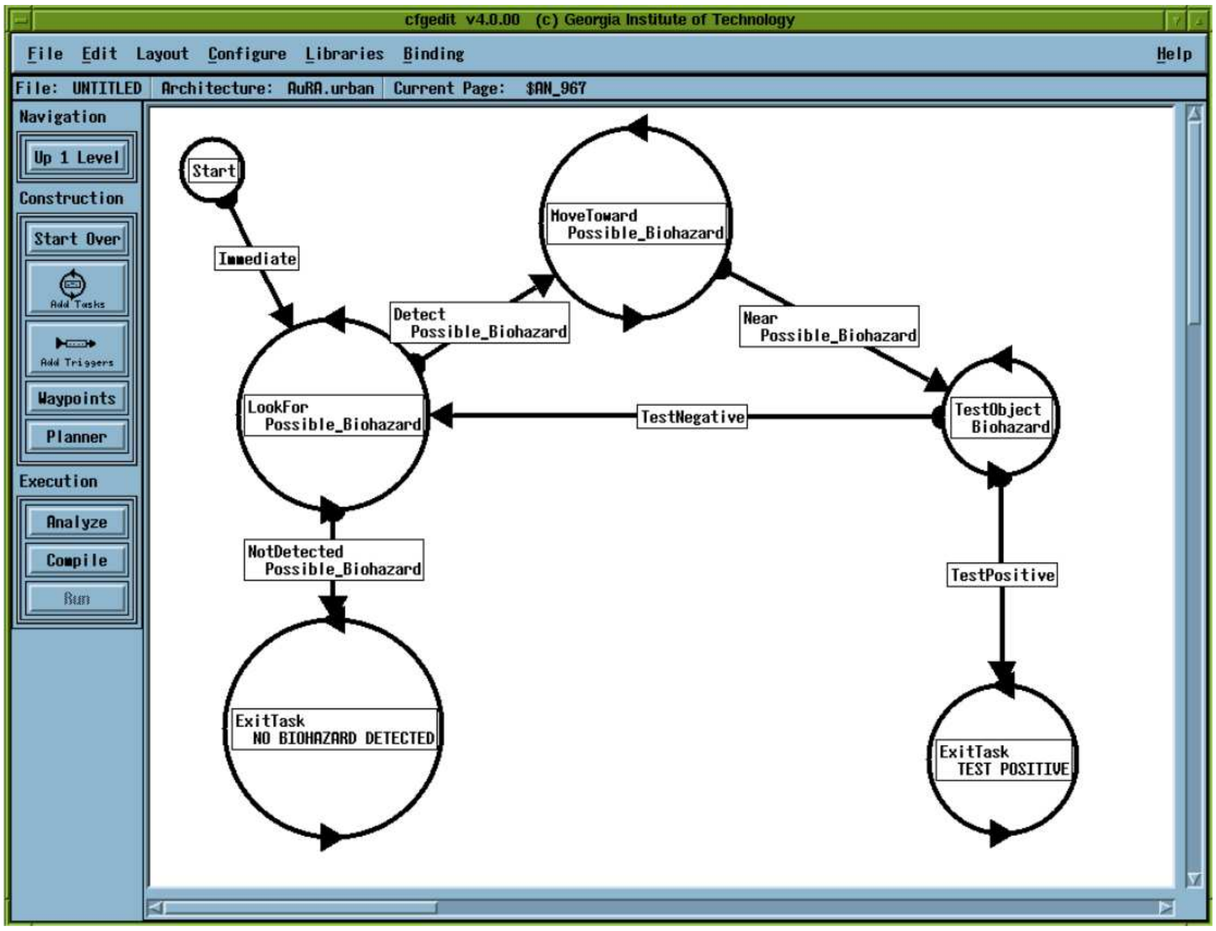

cfgedit v4.0.00 (c) Georgia Institute of Technology
File Edit Layout Configure Libraries Binding Help
File: UNTITLED Architecture: AuRA.urban Current Page: $AN_967
Navigation
Up 1 Level
Construction
Start Over
Add Tasks
Add Triggers
Waypoints
Planner
Execution
Analyze
Compile
Run
Start
Immediate
MoveToward Possible_Biohazard
Detect Possible_Biohazard
Near Possible_Biohazard
LookFor Possible_Biohazard
TestNegative
TestObject Biohazard
NotDetected Possible_Biohazard
TestPositive
ExitTask NO BIOHAZARD DETECTED
ExitTask TEST POSITIVE

Figure 2.11: *MissionLab's Configuration Editor.*

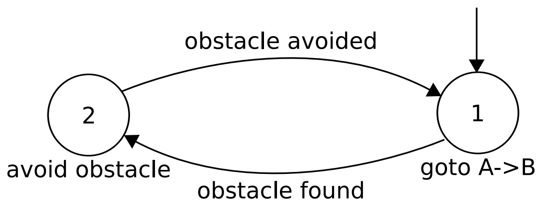


Figure 2.12: *Definition of a schema using 'avoid' and 'fly to' schemas.*

photograph, each with its own parameters. Based on these behaviors the following schemas are defined: takeoff, land, fly to, avoid, detect, recon, shift and main. Each node in a schema may refer to a basic behavior or to another schema. Execution of the different behaviors and schemas works in a purely event-driven fashion.

In this section, some representative examples of mission specification techniques for reconfigurable autonomous vehicles have been described. Some of them make use of text based languages to specify each task, others use graphical representations (such as Petri nets or some form of FSM) or a combination of both. In all cases, the viability of each approach has been demonstrated using both simulated and real vehicles.

While graphical representations are easier to grasp and may facilitate the mission design, its expressiveness is limited and they can easily become too cluttered or hide too many information when details like conditions and messages with their corresponding parameters are thrown in. A text based representation will accommodate these details more easily but it lacks the benefits of a visual representation.

In our system, the language for specifying behavior is State Chart XML (SCXML) (W3C, 2009). 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 (Harel & Politi, 1998). Two important aspects of statecharts are their ability to express hierarchy and concurrency, thus making them appropriate for the description of complex systems. Statecharts are part of the Unified Modeling Language (UML) (OMG, 2010; Booch *et al.* , 2005), which is a widespread graphical modeling language used in industry and academia. Existing UML tools can be used to visually design the mission and then translate the design to SCXML. Other compelling features of SCXML are its scripting and data manipulation facilities, which can be used to specify the vehicle behavior with a fine grain of detail. Being developed as an open standard several implementations already exist that can be used to rapidly prototype our MMa service. Finally, it is worth mentioning that RSML, a language based on statecharts, has been used in the past to formalize system requirements of critical avionics systems (Heimdahl *et al.* , 1998; Thompson *et al.* , 1999).

3 System Architecture

This thesis has been developed as part of the ICARUS Group's effort to push forward UAS technologies for civil applications. The flight and mission management modules presented here are part of a wider set of components organized following the architecture proposed by E.Pastor et al. in (Pastor *et al.* , 2007). This chapter provides an overview of the UAS architecture that accommodates the flight and mission management modules.

3.1 Architecture Overview

The system architecture being developed by the ICARUS Group conceives a UAS as a distributed system, where a number of software components use a common communications infrastructure to exchange information and collaborate. We refer to this software components as services. Each computational node can run one or more services. Communication between services follows a publish/subscribe model and is managed by a middleware layer. There is a collection of services that have been identified as necessary to perform a wide range of missions. These services are standardized by what is called the UAS System Abstraction Layer (USAL).

The USAL concept can be compared to the way operating systems handle device drivers. Computers have hardware devices used for input/output operations, each one having its own particularities. The operating system offers an abstraction layer to access such devices in a uniform way. Basically, it publishes an Application Program Interface (API) that provides end-users with a standardized way to access hardware elements. The USAL makes use of the communication primitives provided by the underlying service-oriented middleware layer.

Another goal of the USAL is to provide a set of components that can be reused across different missions. The available services will cover an important part of the generic functionalities present in many missions. Therefore, in many cases, to adapt the system to a new mission it should be enough to reconfigure the services deployed in the UAS.

3.2 Service-Oriented Middleware

Service Oriented Architectures (SOA) are getting common in several domains. These architectures try to increment interoperability, flexibility and extensibility of the designed system and their individual units by using loosely coupled components. In SOA architectures, functionality is distributed among services and made accessible through well defined protocols.

Following these principles, our system is designed as a network of cooperating services. Services implement the logic of the application and access each other by means of a middleware layer called MAREA (Lopez *et al.* , 2007) that abstracts the execution environment and implements common functionalities and communication channels. When one service needs some externally provided functionality it asks the middleware for the required service. If there is a component that provides the requested functionality, its location is then provided to the client, that will then consume the service. The discovery mechanism is transparently managed by MAREA, that is able to link producers and consumers of data that have no a priori knowledge of their physical location. From the services point of view, the whole system, which comprises both embarked and ground components runs on a single network. The middleware layer also handles all the transfer chores: message addressing, data marshaling and demarshalling, delivery, flow control, retries, etc.

In MAREA, services are offered and consumed following a publish/subscribe model that simplifies programming of distributed applications. Any service can be a publisher, a subscriber, or both simultaneously. There are four kinds of supported communications primitives:

- A *variable* is a structured, and generally short, piece of information offered by a service. This information may be sent at regular intervals or when changes occur.
- An *event* is similar to a *variable* but the middleware guarantees the reliability of the transmission. Events should be used to inform of occasional or important facts.
- *Remote invocations* operate as function calls in a non distributed environment. They represent a classical way to model interactions between distributed components.
- *File transfers* are used for transmitting big chunks of data such as images, video, configuration files, etc.

In a similar fashion to existing avionics buses, such as ARINC 429, MAREA communication primitives identify exchanged data rather than its providers or consumers.

3.3 USAL Services

Previous sections offer a general view of the system architecture and the underlying middleware. In this section we go through the different services that form part of the USAL (Royo *et al.* , 2008), starting with a description of the different categories they fall into. As depicted in Figure 3.1, USAL services are organized in four different categories: Flight, Awareness, Mission and Payload services.

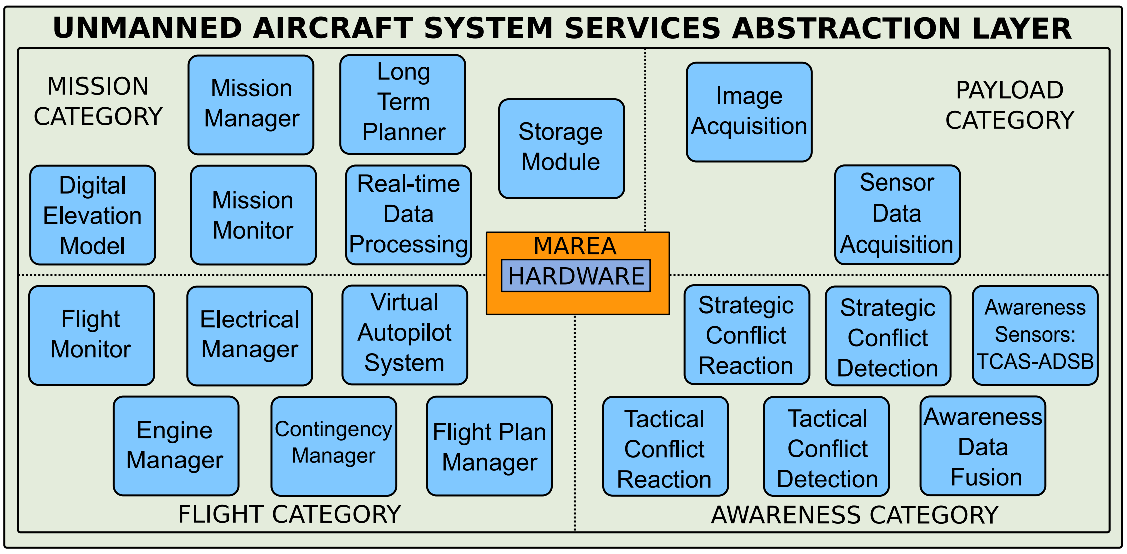


Figure 3.1: *USAL Architecture Global View*

- *Flight Services*: This is arguably the most important category. Not being able to properly control and sustain the UAS' flight will result in a failed mission and put both the platform and third parties at risk. Although not exactly a service, the main element in this category is the autopilot. There are many autopilots in the market and we want to be able to select the best solution for each particular need. For this reason there is a service, the Virtual Autopilot System (VAS), that manages all autopilot interaction details at one end and provides an standardized interface to the rest of services at the other end. Among other functionalities, the VAS supports waypoint navigation primitives. These primitives are then used by the Flight Plan Manager in order to govern the UAS flight. Other services also included in this category such as the Electrical Manager, the Engine Manager and the Contingency Manager help to improve safety and reliability.

- *Mission services*: Mission services are those responsible for the actual execution of the mission. The Mission Manager orchestrates operation of flight and mission related services in order to achieve the mission goals. The MMa listens to system events and responds in a purely reactive fashion. Services that store and analyze sensed data are also found in this category. Planning services will also fall into this category.

- *Payload services*: In this category, those systems that handle operation of sensors and actuators are found. There are many kinds of sensors that we may need to take care of: GPS, IMU, Anemometers, visual, infra-red and radiometric cameras, chemical and temperature sensors, radars, etc. Although there are far less actuators, some examples would consist in flares, parachutes or loom shuttles.

- *Awareness services*: This category includes those services that gather information about the environment the UAS is operating in. These services are critical for a successful integration of UAS in non-segregated airspace. Awareness services handle interaction with cooperative aircrafts through transponders, TCAS or ADS systems and try to detect non-cooperative aircrafts through visual or other kinds sensors. Services in this category will also take control and command emergency maneuvers in critical situations where an immediate response is required.

Although the USAL is composed of a large set services, not all of them need to be present at all times. Only those required for a given configuration/mission should be present and/or

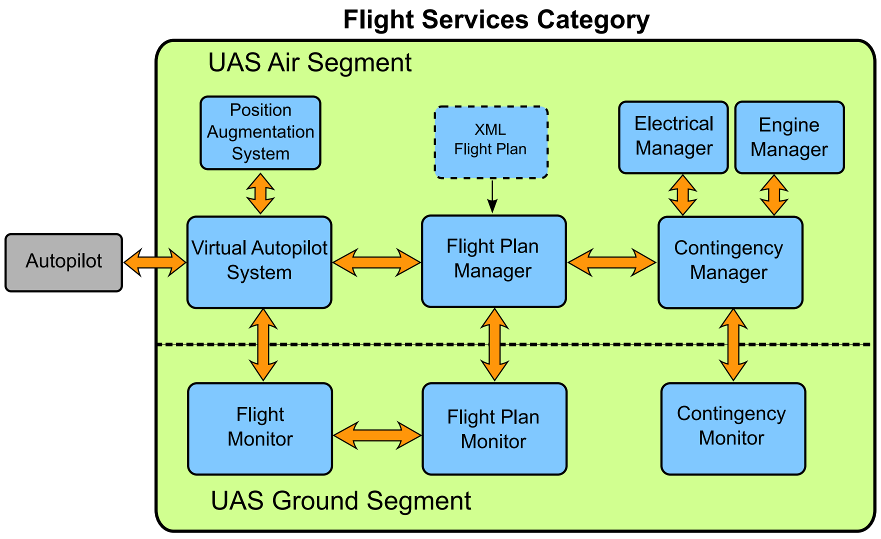


Figure 3.2: *Overview of the Flight Services category*

activated in the UAS.

Next sections go through the main services we can found in each category. It should be noted that the degree of development of the different services greatly varies. Early versions and prototypes of most flight services are already available. Other services, such as the Long Term Planner or services in the Awareness category, should be seen as needs that have been identified but not yet addressed.

3.3.1 Flight Services

There are several goals flight services aim at. With regard to the embarked autopilot, we want to abstract other services from its specific details and also be able to extract information from its internal sensors. We also want to extend its capabilities and provide flight-plan definition mechanisms that improve by large what is found in current commercial solutions. Finally, another goal consists in monitoring operation of the power system and the UAS engine and responding to detected contingencies. Services in the flight category (see Figure 3.2) are responsible for providing these capabilities. A brief description of each service follows:

- The *Virtual Autopilot System* interacts with the selected autopilot and therefore needs to be adapted to its peculiarities. The VAS offers a common and well-defined interface to all services that require access to autopilot capabilities. This includes, although it is not limited to, waypoint based navigation and access to the autopilot telemetry data.
- The *Flight Plan Manager* is a service designed to provide flight-plan capabilities that go beyond simple sequences of waypoints. The FPM provides structured flight-plan phases with built-in emergency alternatives, leg based navigation and constructs to enable forking, repetition and generation of complex trajectories from a reduced number of parameters.
- The *Engine Manager* and the *Electrical Manager* are respectively in charge of monitoring engine and electrical parameters and detect problems in these subsystems. Additionally, both services are able to estimate the remaining time for performing the mission in nominal conditions.
- The *Contingency Manager* collects status information from multiple sources: engine, electrical, fuel, communications, etc. to determine if a contingency occurs and decide

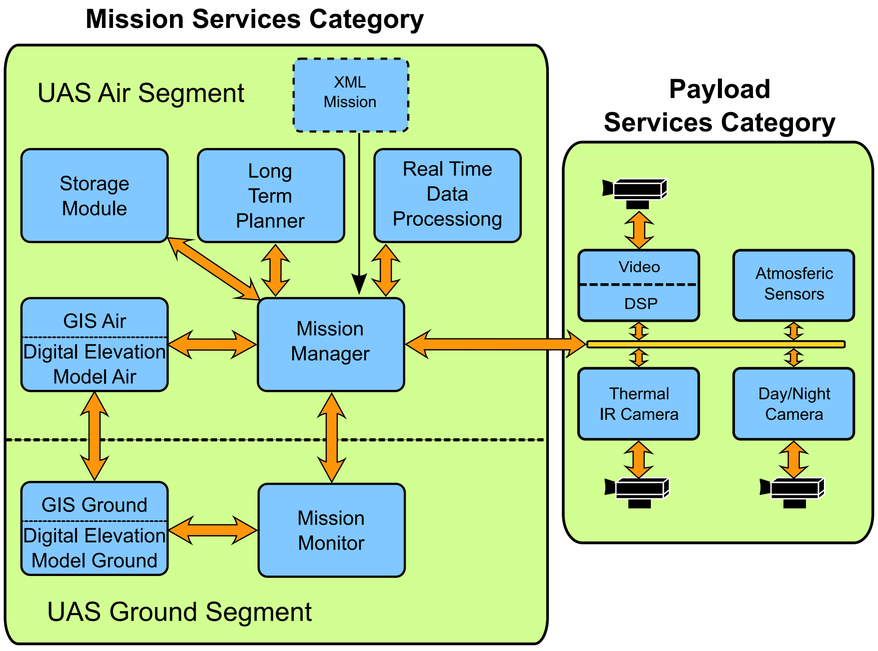


Figure 3.3: *Overview of the Mission and Payload Services category*

what type of reaction is required. The response may range from continuing operation in a degraded mode to activating a flight termination system for the immediate finalization of the mission.

- The *Flight Monitor*, *Flight Plan Monitor* and *Contingency Monitor* listed as ground segment services provide the consoles for supervisory control of their respective embarked counterparts.

3.3.2 Mission Services

The USAL Mission category offers a number of predefined services to implement a wide range of missions, namely the *Mission Manager*, the *Real-Time Data Processing*, the *Storage Module*, the *GIS/DEM Database* and *Mission Monitor*. Figure 3.3 shows its fundamental components as well as their relations.

- The *Mission Manager* orchestrates operation of the USAL services. The behavior of the UAS during the mission is defined by means of an XML representation of Statecharts. During execution of the mission, the MMa listens to events coming from other services and reactively responds according to the current state. In this way, the MMa is going to active those services that should be running at a given time, to modify the flight plan or to change how a given service is operating.

- The *Real-Time Data Processing* will be able to extract information from raw data and pass it on to other services. It will offer image processing operations to allow the MMa and other subscribed services to detect relevant mission events, e.g., that a potential object of interest has been discovered.

- The *Mission Monitor* enables end-users to supervise the evolution of the mission and provide support to decision makers. For example, during a wild land fire monitoring mission, information regarding the current state of the fire would be displayed.

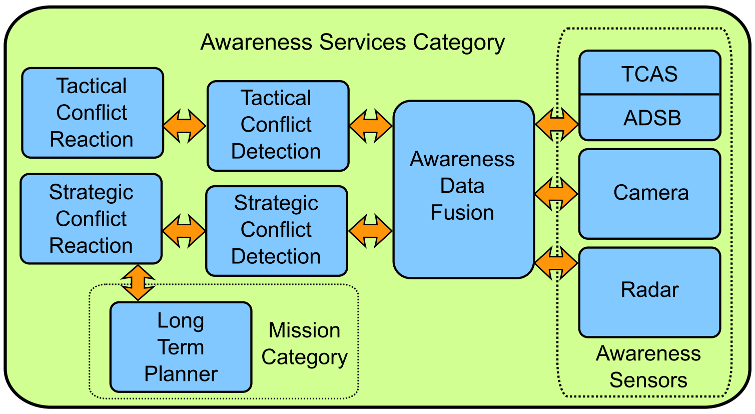

Awareness Services Category
Tactical Conflict Reaction
Tactical Conflict Detection
Strategic Conflict Reaction
Strategic Conflict Detection
Awareness Data Fusion
TCAS
ADSB
Camera
Radar
Awareness Sensors
Long Term Planner
Mission Category

Figure 3.4: *Overview of the Awareness Services category*

- The *GIS/DEM* services provide the system with information derived from digital elevation models and other geographical information sources. A light-weight service is embarked on the UAS platform while a more heavy-duty service is placed on the ground segment with less restrictions on available resources. While this services has been placed in the Mission category there is a clear overlap with Awareness.
- The *Long Term Planner* service is responsible for making decisions about the future trajectory of the aircraft. This service could, for instance, compute an alternative route to solve a conflict detected by one of the awareness services in a situation where immediate response is not required.
- The *Storage Module* provides easy access to the storage medium on-board (compact flash, hard disk, etc.). It stores the data generated by different sensors: camera, telemetry, etc. and can also be used to save service configurations or even backups of the deployed software components. Of course, it plays an important role in situations with limited down-link bandwidth, keeping more information than can possibly be sent to the ground segment.

3.3.3 Payload Services

Carrying payload is the ultimate reason for having an UAS, and often times constitutes the most expensive equipment. Payload includes cameras that operate on different spectrums, radar sensors, atmospheric and chemical sensing devices, etc. Payload services are defined to provide a friendly interface and control operation of these raw data acquisition sensors.

3.3.4 Awareness Services

An UAS is a highly instrumented aircraft that has no pilot on board. When performing remote sensing missions the possibility of intercepting other aircrafts, which may operate under Visual Flight Rules and lack the equipment to actively broadcast its position, must be considered. Therefore, the UAS must be able to transmit enough information to keep an on ground pilot with an adequate level of awareness or implement equivalent capabilities. Awareness services monitor the surrounding environment and take control of flight management in conflict situations. When such thing occurs, flight and mission services recover its role once conditions become normal. The services in the Awareness category are shown in Figure 3.4. A brief description of each one follows:

- The *Awareness Data Fusion* service is designed to collect all available data about air vehicles surrounding our UAS plus terrain and meteorological conditions. All this information can be obtained either by on board sensors or through an external provider.

- The *Tactical/Strategic Conflict Detection* services will analyze the fused information in order to detect potential collision conflicts with objects/terrain/bad climate. Depending on the type of conflict, different types of reaction procedures will be activated. While reaction is executed it will keep monitoring than the conflict is really being avoided.

- The *Tactical/Strategic Reaction* services, will implement avoidance procedures according to the severity of the conflict. Tactical reaction is designed in a way that it can overtake the Flight Plan Manager in order to execute an aggressive maneuver. Once completed, the FPM will regain control. An strategic reaction will command the FPM to slightly modify its selected flight plan trying to avoid the conflict but at the same time retaining the original mission.

- A set of dedicated *awareness sensors* will gather information relative to possibly conflicting collaborative and non-collaborative aircrafts.

3.4 Conclusion

In this chapter, the set of services that we envision as forming part of the UAS architecture has been presented. This information provides the background to understand the environment in which the Flight Plan Manager and the Mission Manager will run. The architecture is characterized by being highly distributed and very flexible. Not all services are required for the UAS to operate and they can be added, removed or changed depending on the mission needs. There are two elements that facilitate this high degree of flexibility: (1) a middleware layer, that abstracts services from the networking and low level communication details, and (2) a service abstraction layer, that defines the interfaces services must conform to.

Implementation of the architecture is a work in progress. In this dissertation we focus on the Flight Plan Manager and the Mission Manager: two services that play a key role in governing the UAS flight and operation of the mission payload. The following chapters address, in detail, all their different aspects.

4

Flight Plan Specification Language

This chapter presents the specification language proposed to design UAS flight plans. Section 4.1 explains the structure and the different elements that can be found in a flight plan. Support for emergency plans is discussed in Section 4.2. In Section 4.3, we show how to dynamically adapt the flight plan to mission needs during its execution.

4.1 Base Flight Plan

Most current UAS autopilot systems rely on lists of waypoints as the mechanism for flight plan specification and execution (Chao *et al.* , 2007). 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 adapt to mission circumstances must be introduced. For that reason a new flight plan specification mechanism is proposed.

Some of the ideas that the flight plan specification language is based on come from current practices in commercial aviation for the specification of RNAV (FAA, 2008) procedures. As seen in Chapter 2, 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 adaptability is increased by means of parametric legs.

The flight plan is stored in an XML document that will be submitted to the UAS in order to carry out its execution. Following sections describe the contents of the XML flight plan specification document.

4.1.1 Flight Plan Document Structure

The root node of the XML document that contains the UAS flight plan is *FlightPlan*. Listing 4.1 shows the elements contained within the root element. To make XML listings more readable some content has been replaced by ellipses.

Listing 4.1: XML flight plan document structure.

```
<FlightPlan xmlns='http://icarus.upc.es/schema/FlightPlan/1.1'>
  <Locale> ... </Locale>                      <!-- units and separators -->
  <Fixes> ...  </Fixes>                       <!-- specific named locations -->
  <EmergencyPlans> ... </EmergencyPlans> <!-- emergency flight plans -->
  <MainFP> ... </MainFP>                      <!-- main flight plan -->
</FlightPlan>
```

Locale specifies which units are used for speed, altitude and distances. It also specifies which 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. Emergency plans and the main flight plan share the same structure, which is presented in Section 4.1.4. Some details specific to emergency plans are discussed in Section 4.2.

4.1.2 Locale Settings

Locale settings specify what units are used for speeds, angles, altitudes and distances. Decimal and group separators are also indicated. Possible values for each one of these elements are shown in Table 4.1:

Table 4.1: *Locale settings supported values.*

speedUnits		angleUnits		altitudeUnits distanceUnits		decimalSeparator	groupSeparator
ms	m/s	deg	degrees	m	meters	in principle it could be any string, but most probably '.' and ','	as in decimalSeparator plus empty
kt	knots	rad	radians	nm	nautical miles		
				ft	feet		

Listing 4.2 shows an example with some possible values. The example states that all speed values included in the flight plan are in meters/second, all altitudes and distances are in meters and a decimal point is used as the decimal separator. An empty *groupSeparator* element indicates

that no thousands separator is used.

Listing 4.2: Locale settings example.

```
<Locale>
   <speedUnits>ms</speedUnits>
   <angleUnits>deg</angleUnits>
   <altitudeUnits>m</altitudeUnits>
   <distanceUnits>m</distanceUnits>
   <decimalSeparator>.</decimalSeparator>
   <groupSeparator/>
</Locale>
```

4.1.3 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 4.3, a fix has an identifier, a name and a description followed by its latitude and longitude coordinates.

Listing 4.3: 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>
```

Fixes are closely related to waypoints, which designate 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, they indicate the required speed and altitude of the aircraft at that waypoint. Fixes and waypoints are used to specify the destination of higher level leg constructs. Listing 4.4 shows an example of a waypoint that refers to a fix. The *dest* element is part of a leg specification and is described in section 4.1.6.

Listing 4.4: XML description of a waypoint.

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

Table 4.2 below describes the data type and optionality of each waypoint element.

4.1.4 Main Flight Plan

A flight plan specifies the path followed by the aircraft. As seen in Figure 4.1, each flight plan is composed of a sequence of stages, such as take-off, departure procedure and others, which must

Table 4.2: *Data types for waypoint elements.*

Element	Data Type	Optionality
coordinates	lat lon in two possible formats: dd°mm'ss.ss"N|S dd°mm'ss.ss"E|W or two real values	Either fix or coordinates are required
fix	fix id	"
fly-over	bool	Optional (default value is false)
altitude	double	Optional
speed	double	Optional

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.

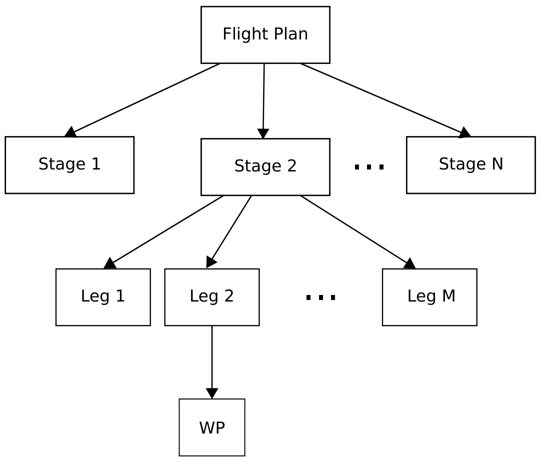


Figure 4.1: *A flight plan is composed of stages, legs and waypoints*

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 4.5). Optionally, for the main flight plan a list of emergency plans can be specified. This provides default emergency plans that can be superseded by emergency plans specified at stage or leg levels.

Listing 4.5: 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>EmergencyFP1 EmergencyFP2 ... </emergency>
</MainFP>
```

4.1.5 Stages

Stages organize legs and constitute high-level building blocks for flight plan specification. Each stage corresponds to a conceptually well defined flight phase. A stage groups together a collection of legs that seek a common purpose. Stages must comply with the following rules:

- Every stage, except for the first and last stages, has a single predecessor and a single successor.
- Stages are always flown in sequential order.
- A stage may have more than one exit leg. E.g., a take off procedure may end at different points depending on the selected take off direction.
- A stage may have more than one entry leg. E.g., a departure procedure, that follows a take-off, can start at different positions.
- 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 if necessary.
- Emergency flight plans are an exception to the previous rule. Since the first stage of an emergency plan may have more than one initial leg the selected leg to enter the emergency flight plan will be the one whose destination is closest to the current aircraft position.
- The correspondence between the exit legs of an stage and entry legs of the next stage is determined by their position in the respective *finalLegs* and *initialLegs* lists.
- All reachable legs must have either a next leg in the same stage or must appear in the *finalLegs* list. In other words, we must make sure that the aircraft cannot reach a dead end.

Listing 4.6 shows what elements can be found inside a stage. It should be noted that we are currently focusing on in-flight leg based navigation. Stages like *Taxi*, *TakeOff* and *Land* (see Table 4.3) can be seen as placeholders that will eventually contain whatever information is required by the VAS to perform them in an automatic mode, once this capabilities become available.

Listing 4.6: 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> ... </emergency>           <!-- Emergency flight plans          -->
</stage>
```

Each stage has an identifier, a name and an optional description. Its purpose is specified using the *type* attribute. The *manualOnly* attribute will be set to true if automatic execution of this stage is not possible, e.g., when taxiing. When a stage marked as such is encountered it is responsibility of an on-ground human pilot to control the aircraft. Valid values for the *type* attribute can be found in Table 4.3.

The *legs* element lists all the legs that are part of this stage. Additional elements are *initialLegs* and *finalLegs*, which are white space separated lists indicating what are the sets of initial and final legs of the stage respectively.

Table 4.3: *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.

Stages have an optional element indicating which emergency flight plans are to be carried out when an emergency occurs. This emergency plans will lead to a near area where landing is possible. If other emergency plans are specified at leg level, the latter ones will prevail.

4.1.6 Legs

A leg specifies the flight path to get to a given waypoint. In general, legs contain a destination waypoint and a reference to the next leg. The *dest* element specifies which is the destination waypoint. Next and previous legs are indicated respectively by the *next* and *prev* elements. Only intersection legs, which mark decision points, are allowed to specify more than one next and previous legs.

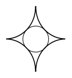
There are four different kinds of legs:

- 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.

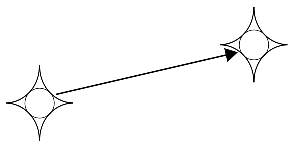
Intersection legs differ from the rest in that they may be reached from more than one predecessor and may lead to more than one successor. All legs may include an optional parameter indicating what are the emergency flight plans available when a contingency occurs during the execution of the leg.

4.1.6.1 Basic Legs

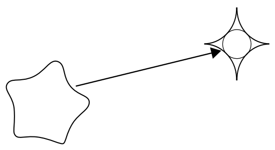
This section describes the basic legs available to the flight plan designer. They are referred to as basic legs to differentiate them from control structures like iterative or intersection legs and parametric legs. All of them are based on already existing ones in RNAV. Its original name is preserved. An schematic view of the different basic legs available is shown in Figure 4.2.



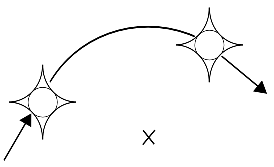
(a) *Initial Fix*



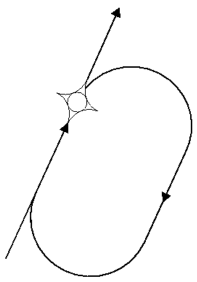
(b) *Track to a Fix*



(c) *Direct to a Fix*



(d) *Radius to a Fix*



(e) *Holding Pattern*

Figure 4.2: *Basic leg types available.*

Initial Fix (IFLeg)

An 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 (TFLeg)

A Track to a Fix corresponds to a straight trajectory from waypoint to waypoint. Initial waypoint is the destination waypoint of the previous leg. Listing 4.7 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, which must be reached at the specified speed.

Listing 4.7: XML description of Track to a Fix leg.

```
<leg id="L1" xsi:type="TFLeg">
   <dest>
      <coordinates>41º17’38.38”N 2º4’35.82”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)

A 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 (RFLeg)

A 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 (*Left* or *Right*).

Listing 4.8: Radius to a Fix leg.

```
<leg id="L2" xsi:type="RFLeg">
   <dest> ... </dest>
   <next>L3</next>
   <center>41º17'38.38"N 2º5'27.49"E</center>
   <direction>Right</direction>
</leg>
```

Holding Pattern

A Holding Pattern specifies a racetrack-like 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 (HCLeg). The holding pattern will be terminated after a given number of iterations or when the condition result is set to 0 (regardless of the number of iterations). Any other value will cause a repeated execution of the holding pattern.

In all cases, *dest* specifies the initial (and final) waypoint of the holding pattern and is located just before the beginning of the first turn. *course* specifies the orientation of the holding pattern in degrees. *turnDir* indicates whether the aircraft turns to the right or to the left. *d1* indicates the distance between the turning centers and must be equal or greater than *d2*, which is the diameter of the turning segments.

Listing 4.9: Holding to a Fix leg.

```
<leg id="holding" xsi:type = "HFLeg">
   <dest> ... </dest>             <!-- Holding fix -->
   <course>110</course>           <!-- Azimuth -->
   <direction>Left</direction>    <!-- Turn direction -->
   <d1>2000</d1>                  <!-- Distance between turn centers -->
   <d2>1600</d2>                  <!-- Turn diameter -->
</leg>
```

Table 4.4 summarizes the required parameters for each basic leg type.

Data types for each one of the previous parameters are as shown in table 4.5:

4.1.6.2 Iterative Legs

Iterative legs are constructs that enable the UAS to exhibit repetitive behavior. An iterative leg groups together a number of legs that will be repeatedly executed. These legs form the body of the iterative leg. An iterative leg has a single entry (i.e. its body can be entered at a single leg), and a single exit. These entry and exit points are identified in the flight plan using the *first* and

Table 4.4: *Parameters for basic leg types.*

	dest	next	prev	turn direction	arc center	course	d1	d2	altitude	climb rate	condition	upper bound	emergency
IF	√	♦	★	-	-	-	-	-	-	-	-	-	O
TF	√	♦	★	-	-	-	-	-	-	-	-	-	O
DF	√	♦	-	-	-	-	-	-	-	-	-	-	O
RF	√	♦	★	√	√	-	-	-	-	-	-	-	O
HA	√	♦	★	√	-	√	√	√	√	√	-	-	O
HF	√	♦	★	√	-	√	√	√	-	-	O	O	O

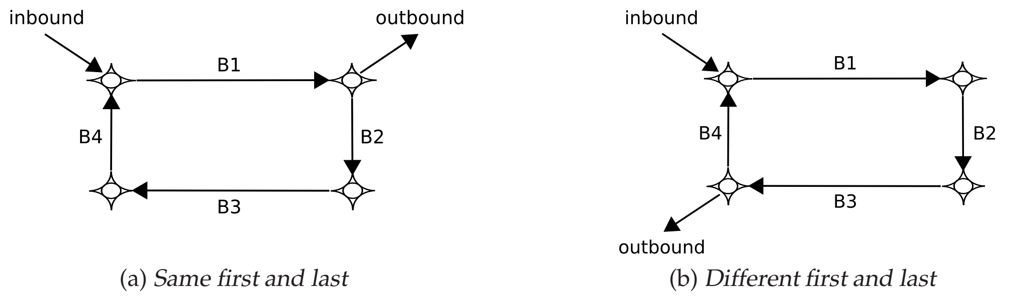
√	Required.
O	Optional.
♦	Required except for last leg of stage.
★	Required only if within back and forth iterative leg.

Table 4.5: *Data types.*

Element	Data type	Element	Data type
dest	waypoint element	d2	double, units as set in *Locale*
next	leg id	altitude	double, "
prev	leg id	climb rate	double in altitude/distance units
turn direction	enumerated (Left or Right)	condition	string id of condition
arc center	coordinates (see waypoint)	upper bound	positive integer
course	double, azimuth in degrees	emergency	id of emergency flight plan
d1	double, units as set in *Locale*		

last tags respectively. The number of iterations and, optionally, a condition can be specified to determine when to leave the iterative leg. Every time the last leg is executed an iteration counter is incremented. When the given count is reached or an specified condition is no longer satisfied the leg will be abandoned proceeding to the next one.

Figure 4.3 shows the structure of an iterative leg. An inbound arrow represents the leg where we come from and an outbound arrow the leg that is going to be executed after the iterative one. Diagrams 4.3a and 4.3b only differ in whether the first and last legs are the same or not.



(a) *Same first and last* (b) *Different first and last*

Figure 4.3: *Iterative leg examples.*

Listing 4.10 shows an XML description corresponding to 4.3a. There are attributes to specify the leg id and an indication of its type. *next* contains the name of the leg to execute after this one. The *body* element contains the list of all legs that form the body of the iterative leg. Since in the example the first and last legs to be executed are the same, *first* and *last* have the same value. *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 4.10: XML description of an iterative leg.

```
<leg id="Loop" xsi:type="IterativeLeg">
   <next>outbound</next>
   <body>B1 B2 B3 B4</body>        <!-- Body of the loop        -->
   <first>B1</first>               <!-- First body leg          -->
   <last>B1</last>                 <!-- Last leg before exiting -->
   <upperBound>5</upperBound>      <!-- Repeat five times       -->
   <cond>CondID</cond>             <!-- Condition that controls termination -->
</leg>
```

4.1.6.3 Intersection Legs

Intersection legs mark points where two or more different paths meet and where decisions on what to do next can be made. All converging and diverging paths will respectively end and start at an intersection leg. An intersection leg contains a list of next legs and a condition id that is used to select one of them. The role of *next* element is to identify the default leg. If present, this default leg will be taken unless the condition result says otherwise. If not present, the aircraft waits performing a holding pattern until the condition outcome becomes known. The integer value returned by the condition will be used as an index to select the next leg from the list of specified possibilities. The use of intersection legs to specify iterative behavior is not allowed.

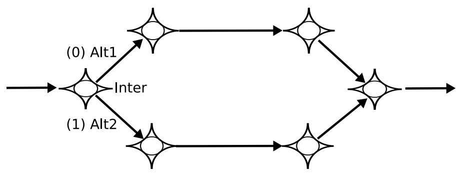


Figure 4.4: *Intersection leg example.*

Figure 4.4 illustrates a situation where two intersection legs are used. Listing 4.11 shows the XML of the first one.

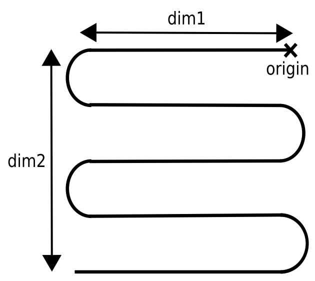
Listing 4.11: XML description of an intersection leg.

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

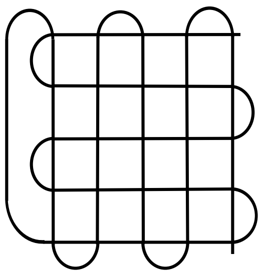
4.1.6.4 Parametric Legs

Previous language elements already provide a powerful mechanism for specifying complex trajectories, but there will be situations where the specification of UAS maneuvers requires a long list of legs, e.g., when performing an scanning pattern over a region of interest. To handle these situations more efficiently parametric legs are introduced. With parametric legs the flight path is automatically generated from a reduced number of inputs. If the mission goal is to systematically explore a given area, instead of writing down the complete UAS trajectory, we can provide the parameters that determine the geometry of the area and all the legs necessary for its exploration will be automatically generated. This approach has an important benefit: should the area of interest change, we can easily recompute the whole UAS trajectory just by updating some parameters.

As an example, Figure 4.5 shows some possible patterns for exploring a given area, as in 4.5a and 4.5b, or a more specific point, as in 4.5c. In all these situations we can benefit from the use of parametric legs. Eventually a library of different parametric legs will be available complete enough so that a wide range of missions can be performed.



(a) *Basic scan pattern*


(b) *Complex scan pattern*

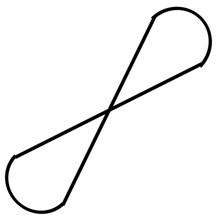

(c) *Scan point pattern*

Figure 4.5: *Scanning patterns.*

Listing 4.12 shows the XML description of the parametric leg seen in figure 4.5a. The *dest*, dimensions and *angle* elements determine the geometry of a rectangular area. Separation indicates the distance between each pass. The initial values given in the specification can be updated during the flight by the UAS operator or an automated mission control service. When one of these parameters changes the flight path will be recomputed.

Listing 4.12: XML description of a parametric leg.

```
<leg id="missleg" xsi:type="BasicScanLeg">
   <dest>
      <coordinates>
         41.5493424917977 1.77254310685181
      </coordinates>
      <speed>60</speed>
   </dest>
   <dim1>6000</dim1>
   <dim2>5500</dim2>
   <angle>80</angle>
   <separation>800</separation>
</leg>
```

4.1.7 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 Flight Plan Manager (see Chapter 5) know when to leave the current leg and proceed to the next one.

From a flight plan perspective, conditions can be seen as <key, value> pairs, i.e. a string id with an associated value. Each leg that depends on a condition contains the corresponding id and the system will be able to obtain its associated value and act accordingly. When the value of a condition is modified the Flight Plan Manager recomputes all affected waypoints. Changes to the condition value may be performed by a human operator or other systems that interact with the Flight Plan Manager service. No restrictions are put on what the conditions represent, they could be based on elapsed flight time, on the completion of a given task, on some payload event or parameter threshold, etc.

4.2 Emergency Flight Plans

All flights require a single main flight plan, however additional emergency flight plans may be present. The main difference between the main flight plan and emergency plans is that while the main plan includes the whole set of stages, emergency plans only cover the finishing stages of a flight. The reason for not including all possible stages in an emergency plan is that they only get executed when something goes wrong during the mission, i.e. when the aircraft is already flying.

Another important characteristic that differentiates them from the main plan is that a higher degree of determinism is required. The inclusion of iterative and intersection legs in the main flight plan makes the total execution time difficult to predict. To address this issue iterative legs are not allowed inside emergency plans. Intersection legs are allowed as long as a default path is set. If any holding patterns appear in an emergency plan their number of iterations must be set to zero. These restrictions provide a bounded default path but still allow some degree of flexibility for a on-ground operator to make final adjustments. In the specification of the emergency flight plan time, estimations for the default and the more time consuming path will be provided.

As seen in Listing 4.13, the structure of an emergency flight plan is the same as in the main flight plan. The *defaultTime* and *maxTime* attributes provide an estimation of the required time for executing the default path and the longest one. Both values are in seconds. After a name and a description comes the list of stages indicating how to proceed to reach the landing site of choice. Another difference with regard to the main flight plan is that an emergency plan does not contain emergency alternatives.

Listing 4.13: Emergency flight plan structure.

```
<EmergencyFP id="FPID" defaultTime="1800" maxTime="2700">
   <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>
</EmergencyFP>
```

Having multiple flight plans (the main one and emergency alternatives) in the same document raises the question of what to do when one of the stages could be used in more than one plan. In this case, it is not necessary to replicate the stage code at each occurrence within the document. The first occurrence of the stage will contain its code and from there on it can be referenced when needed using the *targetId* attribute. When doing so the *intialLegs* and *finalLegs* lists must be restated, see Listing 4.14.

Listing 4.14: Reusing a previously defined stage.

```
<stage targetId="STID">
   <initialLegs>entry1 entry2</initialLegs>
   <finalLegs>end</finalLegs>
</stage>
```

All rules defined in section 4.1.5 must hold in all places where the stage is used. Note that while we really want to make sure that no dead ends can be reached this still allows us to have unused entries to a given stage. This enables us to define some terminal operations in one emergency plan and reuse them in another one using different entry points.

4.2.0.1 Order of Preference

An important consideration to bear in mind when specifying emergency plans for the different main flight plan elements is that order is relevant. When defining the list of emergency plans available for a given leg or stage, the first emergency plan appearing in the list is considered the preferred one. This preference may be due, for instance, to the conditions of the landing site. It may be the case though, that after detecting a contingency situation, the available flight time is not sufficient to follow the preferred emergency route. That's why we allow having more than one emergency plan and, also, why being able to estimate the required flight time for executing each emergency plan is needed.

Emergency alternatives can be defined at flight plan, stage or leg level, but only those set at the lower level of the hierarchy will be considered. For example, if one emergency plan is set at stage level and another one is set for a given leg within the stage, only the latter will be taken into account. Emergency alternatives specified at the stage level will only be considered when flying legs that do not specify other alternatives.

4.3 Flight Plan Updates

There are two types of modifications that can be applied to the flight plan. The first type consists in setting a new value for a given condition. This can easily be done by sending a message to the Flight Plan Manager that contains both the condition identifier and the new value. The second type of modification consists in using an XML document that resembles the one used to specify the flight plan and provides a description of the desired changes. This section discusses how this second form of updates can be used to tailor a flight plan that has already been submitted to the Flight Plan Manager.

Table 4.6 summarizes what kind of operations, shown on the first column, can be applied to the different flight plan elements. The rationale behind the availability of the different operations follows:

- For consistency locale settings defined in the initial flight plan cannot be changed. All modifications to the flight plan must follow the original settings.
- Fixes, which are points of interest relevant to the mission, can be added at any time, but they cannot be removed nor modified. Its removal is not allowed because some other part of the flight plan could depend on them. A fix cannot be modified because if its attributes where to change during the mission then it really should not be a fix in the first place.
- Emergency flight plans can be added and changed. Again we do not want to delete something that could be referenced somewhere else. While emergency plans should not

suffer significant changes during the mission it is certainly conceivable for the UAS operator to want to make some adjustments.

- The main plan can be changed for a better adaptation to the ongoing mission. It is assumed that a significant effort has been put into the planning of the mission before it is actually started. Replacing the main plan as an afterthought is not allowed.
- There is a fixed set of stages that must be executed in sequential order (see Section 4.1.5). It is not possible to add or remove additional stages. Stages can change as a result of modifications occurring at leg level.
- In its simplest form, a change to a leg will consist in updating some of its parameters, e.g., the coordinates of the destination waypoint. More complex situations may involve dynamically adding and removing legs during the mission. This could occur when, for instance, we need to add a new region of interest to a surveillance mission or remove one that does not need further inspection.

Table 4.6: *Supported flight plan updates.*

	Locale Setting	Fix	Emergency Plan	Main Plan	Stage	Leg
Change	-	-	√	√	√	√
Add	-	√	√	-	-	√
Delete	-	-	-	-	-	√

In order to keep the flight plan consistency all changes enclosed in an update message shall be treated as an atomic transaction that either successfully completes or is entirely discarded. Updates do not affect the leg being flown at the time the update message is processed. If the leg being flown happens to be inside an iterative construct then changes will apply to its forthcoming instantiations.

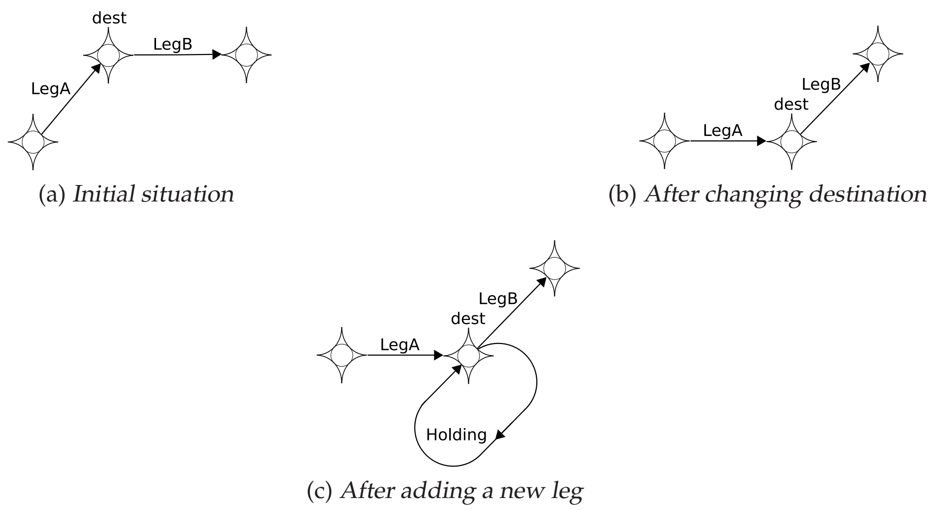
Listing 4.15 shows a simple example illustrating how updates are encoded. The example states that changes are to be applied to elements contained in the *MainFP* section. The only change specified in this message is to update the destination coordinates of a leg called *LegA* to be found inside *MyMissionStage* stage.

Listing 4.15: Example of simple update message.

```
<fpu:FlightPlanUpdate xmlns:fpu='http://icarus.upc.es/schema/FlightPlanUpdate/1.1'
    xmlns:fp='http://icarus.upc.es/schema/FlightPlan/1.1'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:schemaLocation='http://icarus.upc.es/schema/FlightPlanUpdate/1.1'>

<Change>
   <MainFP targetId="FPID">
      <stages>
         <stage targetId="MyMissionStage">
            <legs>
               <leg targetId="LegA" xsi:type="fp:DFLeg">
                  <dest>
                     <coordinates>41.580203095 1.7369781057</coordinates>
                  </dest>
               </leg>
            </legs>
         </stage>
      </stages>
   </MainFP>
</Change>
</fpu:FlightPlanUpdate>
```

Figures 4.6a and 4.6b show how an update such as the one in Listing 4.15 affects the flight plan. After displacing the destination waypoint both the trajectory of *LegA* and *LegB* varies.



(a) *Initial situation* (b) *After changing destination* (c) *After adding a new leg*

Figure 4.6: *Applying updates to the flight plan.*

A more complex example is shown in Listing 4.16, where a holding pattern is inserted between *LegA* and *LegB*. First, the new leg (*Holding*) is added to the flight plan with *LegB* as its destination. Afterwards, *LegA* is changed to reflect that its next leg is no longer *LegB* but the new *Holding* leg. Figure 4.6c graphically shows the result of adding this new leg to our example.

Listing 4.16: Update message with leg insertion.

```
<fpu:FlightPlanUpdate xmlns:fpu='http://icarus.upc.es/schema/FlightPlanUpdate/1.1'
    xmlns:fp='http://icarus.upc.es/schema/FlightPlan/1.1'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:schemaLocation='http://icarus.upc.es/schema/FlightPlanUpdate/1.1'>

<Add>
   <MainFP targetId="FPID">
      <stages>
         <stage targetId="MyMissionStage">
            <legs>
               <leg id="Holding" xsi:type = "HFLeg">
                  <dest>
                     <coordinates>41.580203095 1.7369781057</coordinates>
                  </dest>
                  <next>LegB</next>
                  <course>47</course>
                  <d1>2000</d1>
                  <d2>1600</d2>
               </leg>
            </legs>
         </stage>
      </stages>
   </MainFP>
</Add>
<Change>
   <MainFP targetId="FPID">
      <stages>
         <stage targetId="MyMissionStage">
            <legs>
               <leg targetId="LegA">
                  <next>holding</next>
               </leg>
            </legs>
         </stage>
      </stages>
   </MainFP>
</Change>
```

```
</FlightPlan>
```

Changes to a flight plan can have an impact on its default and maximum expected execution times. New values for these estimations should be included in the update messages if necessary.

4.4 Conclusion

In this chapter the proposed language for specifying UAS flight plans has been detailed. There are several aspects that have been taken into account in its design. The proposed specification language tries to overcome the limitations that dealing with large lists of waypoints in a dynamic environment would impose. To do so, the language relies on legs as the main unit for describing flight plans. Supported legs are borrowed from commercial aviation practices, in particular RNAV, so that the resulting flight plans are expressed in terms familiar to current airspace users. For a better adaption to UAS needs, RNAV legs are extended in a way that allows expressing repetition and choice. Besides that, parametric legs can be used to generate complex maneuvers from a reduced amount of inputs. An update mechanism that enables dynamic adaption to the changing needs of an ongoing mission is also provided. Finally, the proposed language structures the flight plan in different stages and enables the possibility of including alternatives for emergency situations.

As a result, we obtain a very flexible specification mechanism that can accommodate different use cases. Used in a conservative way, the language can be used to specify a very linear and predictable flight path. As we make use of more of the available constructs, such as iterative legs, intersection legs, etc. the ability to adapt to mission circumstances increases, which may have the side effect of making the flight path less predictable. Taken to the extreme, a highly autonomous platform could be dynamically updating the flight plan based on built-in reactive and deliberative capabilities. Since the flight plan is organized in different stages, it is perfectly possible to be very conservative in some of them, e.g., during terminal procedures, and very aggressive in others, e.g., during a mission stage taking place in segregated airspace. This approach could represent a first step in reconciling UAS needs and Air Traffic Management requirements.

5 The Flight Plan Manager Service

Previous chapter described the language used for specifying flight plans. This chapter presents the Flight Plan Manager (FPM), which is the service responsible for their processing and execution. The FPM forms part of a wider ecosystem of services that, together, provide the UAS with all its capabilities. The FPM collaborates with some of those services to perform the execution of the flight plan.

The FPM can be seen as a translator of legs to waypoints. This translation process enables leg based navigation on systems that only support waypoint navigation, which is what most COTS flight control systems offer today. In order to avoid dependence on a specific product, the FPM does not directly interact with the autopilot but with an intermediate service called the Virtual Autopilot System (VAS). The VAS handles the details of the installed autopilot while offering an standard interface to the rest of UAS services. Figure 5.1 shows the hierarchical relationship between the FPM, the VAS and the onboard Flight Control System.

From the VAS or autopilot perspective, the FPM can be seen as a provider of waypoints to fly to. From a mission related services perspective, the FPM is the service to talk to in order to control the flight progress and make it adapt to the mission needs. There are multiple possibilities of interaction with the FPM, the primary ones being setting condition values, sending updates to flight plan elements and triggering execution of emergency plans.

Section 5.1 provides a general description of the FPM operation and its capabilities. Next, in Section 5.2 we discuss how this service interacts with the VAS in order to execute the flight plan. The rest of sections found in this chapter provide details on the internal workings of the service and how it deals with different leg types and user commands.

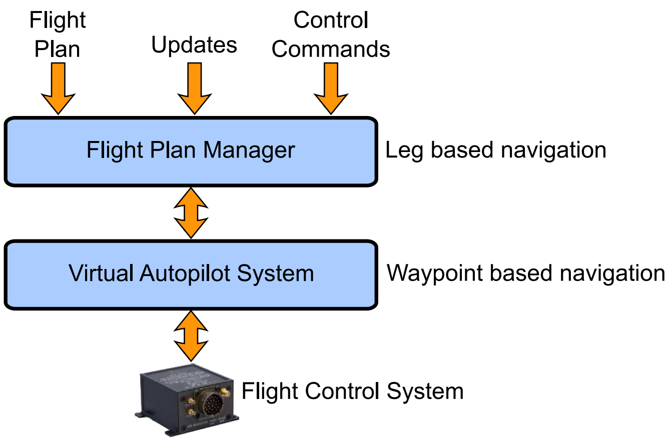
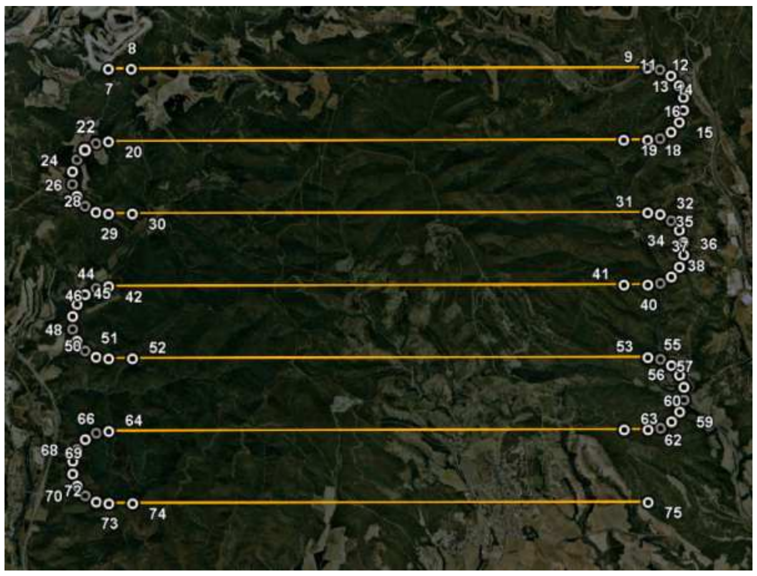
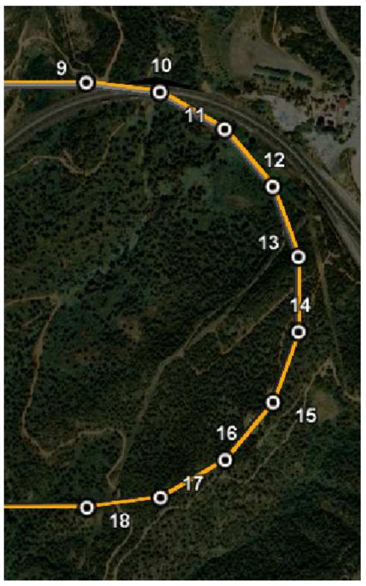


Figure 5.1: *Relationship between FPM, VAS and Flight Control System.*



(a) *Complete maneuver*



(b) *Turn detail*

Figure 5.2: *Example showing generated waypoints for a scanning pattern.*

5.1 Service Description

This section describes the functionality provided by the Flight Plan Manager. As previously stated the main purpose of the Flight Plan Manager consists in processing the submitted flight plan and generating the sequence of waypoints to be flown by the autopilot. As an example, Figure 5.2 shows the waypoints that would be generated for executing a scanning pattern. This maneuver appears in the flight plan as a single leg. During its processing the FPM computes all the waypoints necessary to execute a series of TF legs connected by RF legs. Figure 5.2b shows the detail of a turning maneuver which, as can be seen, is approximated by a sequence of waypoints.

The waypoint generation process can be affected by other services or a UAS operator in order to dynamically adapt the flight to the mission needs. The main requests that must be handled by the FPM are:

- Receive and initiate execution of a flight plan.
- Receive and process updates to the initial flight plan.

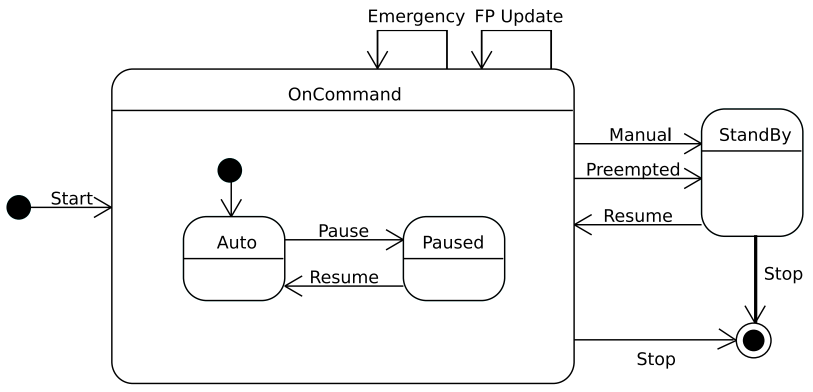


Figure 5.3: *Flight Plan Manager States.*

- Assign new values to conditions that govern selection between alternative routes.
- Skip the leg under execution, i.e. immediately start execution of the next leg.
- Jump directly to a leg located further in the flight path, therefore ignoring some intermediate legs.
- Pause flight plan execution while performing a holding pattern.
- Switch to an standby state, which is going to happen when the UAS is under manual control or controlled by another UAS service.
- Resume operation after a pause or once control is regained.
- Trigger execution of an emergency plan.

All these requirements result in the FPM operating in the states shown in Figure 5.3. In the figure, boxes represent possible states, solid dots represent the default initial ones and arrows indicate transitions between them. There are two main operational states which correspond to the FPM either having or not having control over the flight trajectory. *Auto* and *Paused* are the two possible substates when the FPM is on command. Being in the *Auto* state means that waypoints are generated in order to make progress in the execution of the flight plan. If the FPM transitions to the *Paused* state a message is sent to the VAS to command it to perform a holding pattern while waiting for further instructions. The execution of an emergency plan does not require an additional state. When such a situation occurs the main plan is replaced by an emergency one. The FPM processing the different control inputs and switching between these states requires it to interact with the VAS in order to keep both synchronized.

Apart from the navigation commands sent to the VAS, there are other kinds of messages also generated by the FPM. Other commands related to waypoint management include the FPM requesting the VAS to clear all sent but pending waypoints. This is necessary, for instance, when an emergency occurs. Other types of commands will allow the FPM to change the VAS operation mode to request special operations such as taking-off or landing.

The FPM also generates several information flows that can be exploited by other services. This data includes the position of the aircraft in flight plan terms, i.e. what is the current stage, leg and other leg-related information such as current iteration of an iterative leg. It also periodically publishes what are the emergency plans available, which may depend on the stage or leg being flown, and their estimated duration. Finally, information relative to the current operating state of the service is also provided.

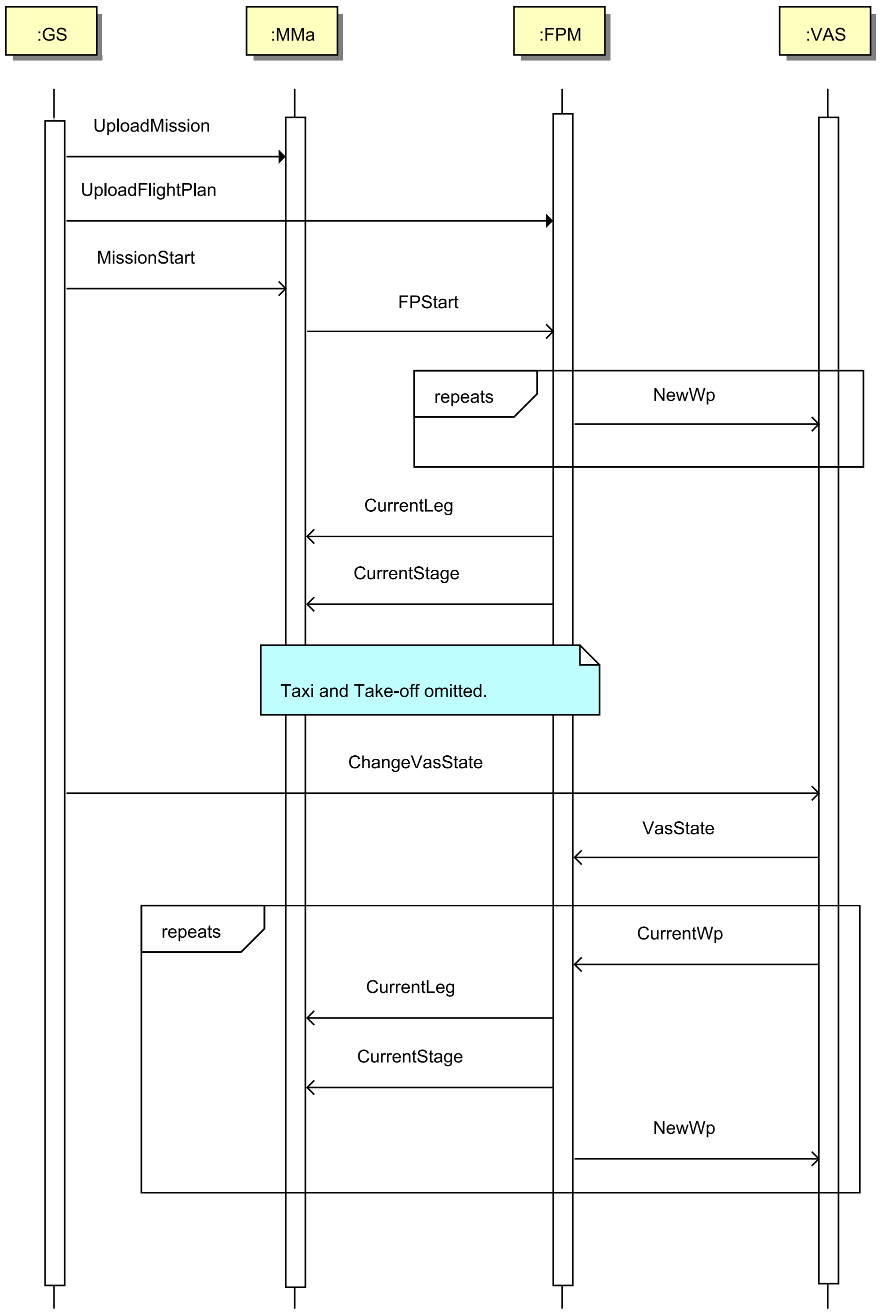


Figure 5.4: *Navigation messages interchanged between FPM and VAS.*

5.2 FPM and VAS Integration

This section describes the main interactions between the FPM and the VAS, which are graphically depicted in Figure 5.4. Table 5.1 summarizes the messages interchanged between both services to perform UAS navigation. In Figure 5.4, GS refers to the Ground Station and MMa to the Mission Manager, a service that is going to be described in later chapters. For the purpose of this section, suffice it to say that the MMa is responsible for coordinating the operation of the UAS payload with the FPM in order to meet the mission goals. Note also that an arrow targeting a given service does not imply other services not being able to receive the message as well.

UAS operation starts by uploading the mission and flight plan definitions to the Mission Manager and the Flight Plan Manager. Mission and payload operation will evolve according to the various phases of the flight.

- *UploadMission*: This function provided by the MMa is used to load a new mission into this service.
- *UploadFlightPlan*: This function is used to load a new flight plan into the FPM service.

As soon as the mission starts, the Flight Plan Manager is also started. At this point, the FPM starts generating waypoints which are progressively sent to the VAS:

- *MissionStart*: Starts mission execution. The SCXML execution engine embedded in the MMa gets started. The execution engine will be driven by events occurring in the system, e.g., notifications that execution of a given leg has started.
- *FPStart*: Start waypoint generation. The flight plan is traversed and waypoints are generated for the legs encountered in the default path. The FPM needs to receive CurrentWp notifications in order to make progress.
- *NewWp*: This event feeds the VAS with the mission waypoints.

Only a limited amount of waypoints is transferred at a time from the FPM to the VAS. This waypoint window is used to ensure that the specified number of waypoints is always available to the VAS. Limiting the amount of waypoints helps keeping communications cost penalties low when old waypoints need to be discarded and replaced by new ones. Such situation may occur due to changes in the flight plan. The initial set of waypoints is immediately transferred, then additional waypoints are sent as the old ones get flown. The VAS informs the FPM and other services by generating an event for each flown waypoint. At the same time, the FPM informs other services (specially the mission management services) of those legs that have been flown. Note that there is not a one to one relationship between waypoints and legs; sometimes a leg has a single waypoint and sometimes a large set of waypoints needs to be flown to complete a leg.

- *CurrentWp*: Indicates the waypoint the system is flying to. This information is offered as an event every time the autopilot switches from one waypoint to the next.
- *CurrentLeg*: As new waypoints are received, the FPM checks whether the leg being flown also changes. If that's the case, a *CurrentLeg* event is generated to inform all subscribed services.
- *CurrentStage*: This event works in a similar way to *CurrentLeg*. In this case, notifications are generated each time the UAS starts flying a new stage.

Waypoint navigation will only start after the VAS switches to the *Navigation* state.

- *ChangeVasState*: Sets the current VAS state.
- *VasState*: Indicates the actual state of the VAS. State is reported each time the VAS switches from one state to another.

The waypoint generation process keeps going until the landing phase, which is directly implemented by the VAS.

Table 5.1: *VAS-FPM interchanged messages during navigation states.*

Protocol Primitives	Name	Composition	Data Type	Range	Unit	Description
Event	NewWp	Latitude Longitude Altitude Speed Fly Over Identifier	Double Double Float Float Boolean USAL Id	0 to 2π rad. 0 to 2π rad. UAS Range UAS Speed N/A N/A	radians radians meters m/s N/A N/A	Submit new waypoint to fly to.
Event	CurrentWp	Wp Identifier	uint (id_ref) uint (id_leg) uint (id_stage)	No range No range No range	N/A N/A N/A	Id of current target waypoint.
Event	ChangeVasState	Target State	Enumerated	N/A	N/A	Set a new VAS State.
Function Event	VasState	Current State	Enumerated	N/A	N/A	Notify the current VAS State
Function	UploadMission	Mission	SCXML	N/A	N/A	Load mission into MMa.
Function	UploadFlightPlan	Flight Plan	XML	N/A	N/A	Load flight plan into FPM.
Event	MissionStart	N/A	N/A	N/A	N/A	Start mission execution engine.
Event	FPStart	N/A	N/A	N/A	N/A	Start waypoint generation engine.
Event	CurrentLeg	NumId Leg Id	uint String	N/A N/A	N/A N/A	Execution of given leg started.
Event	CurrentStage	NumId Stage Id	uint String	N/A N/A	N/A N/A	Execution of given stage started.
Event	UpdateFlightPlan	Update	XML	N/A	N/A	Modify flight plan.
Event	SetCondition	Cond Identifier Value	String uint	N/A No Range	N/A N/A	Set the indicated condition to the given value.
Event	Goto	Leg Id	String	N/A	N/A	Fly directly to the given leg.
Event	Skip	N/A	None	N/A	N/A	Fly directly to next leg.
Event	ClearWps	None	N/A	N/A	N/A	Clear pending waypoints.

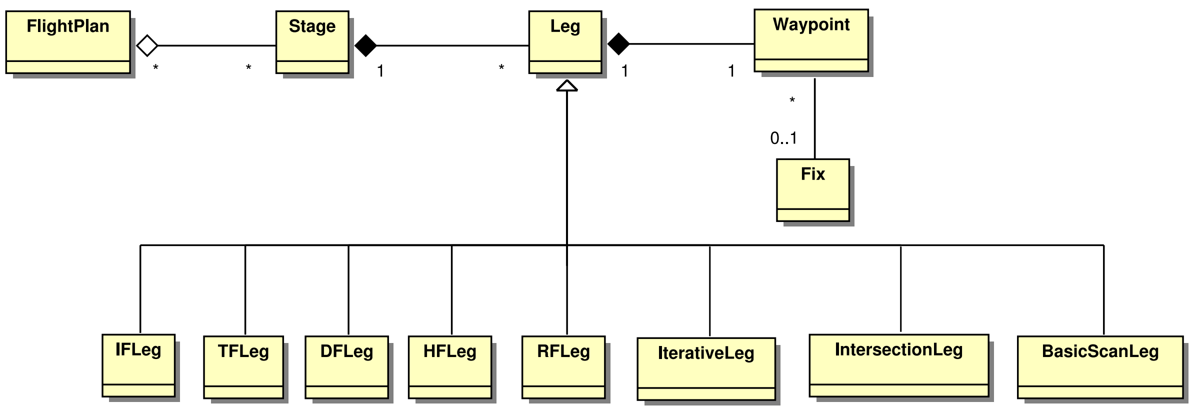


Figure 5.5: *Flight plan main classes.*

5.3 Implementation of the Execution Engine

The flight plan submitted to the FPM is parsed and translated to a internal representation. Figure 5.5 depicts the main classes used to model the flight plan. As seen in the figure, a flight plan has a number of stages that, in turn, contain one or more legs each. These legs can take different forms depending on its type, but all of them have a destination waypoint that can be named or unnamed. If it is a named one, then it is associated to a certain fix.

Flight plan objects are organized forming a tree structure whose root node represents the complete plan (see figure 5.6). Stages are located at the second level with legs following. At this point, some degree of recursion may be found due to iterative legs, whose children legs form the body of the iterative structure. This representation is traversed and waypoints are generated for the encountered legs.

There are two main classes responsible for the FPM behavior, which are the *Controller* and the *WpGenerator*. These two main classes operate following a producer-consumer model:

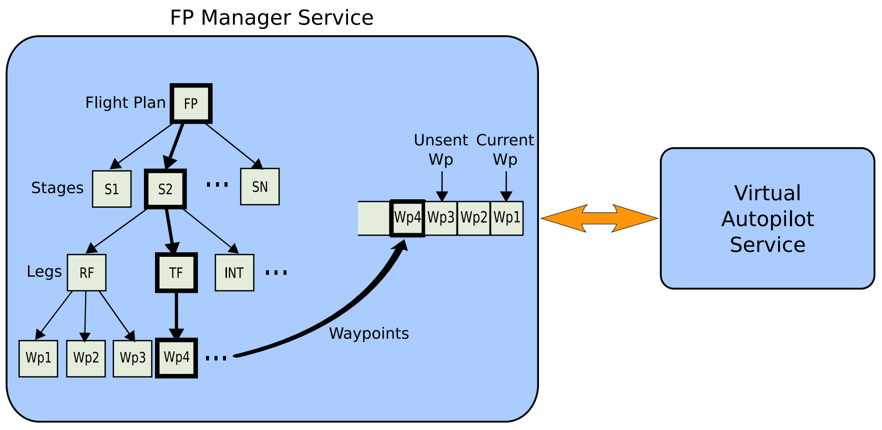


Figure 5.6: *Flight plan execution.*

- The *WpGenerator* has the producer role and generates the flight plan waypoints, which are stored in a queue.
- The *Controller* is responsible for handling interactions of the FPM with other services and managing the states shown in Figure 5.3. One interaction involves taking waypoints from the queue of generated waypoints, therefore, acting as the consumer and sending them to the VAS.

These two classes operate in a decoupled manner: the *WpGenerator* will continually generate waypoints ahead of time until the end of the flight plan is reached, the *Controller*, on the other hand, uses a configurable window size to retrieve generated waypoints and send them to the VAS. Each time a reached notification is received, the *Controller* takes a new waypoint and forwards it to the VAS, thus always ensuring that the VAS holds a minimum amount of wayoints to fly to. The *Controller* also maintains its own queue to keep track of sent but unflown waypoints. The head of this queue contains the waypoint the VAS is heading to. All requests made to the FPM go through the *Controller* who will operate on the internal classes to fulfill them (update the flight plan, modify the value of a given condition, etc).

The additional classes required by the *Controller* and the *WpGenerator* can be grouped in three categories:

- Flight Plan Classes: These maintain the internal representation of the submitted flight plan, including classes for parsing the XML flight plan, initialize data structures and perform updates.
- Waypoint Generation: Each leg type can have one or more classes that can be used to generate waypoints. Having multiple ways of generating waypoints for a single leg type makes it possible to adapt the generation process to the characteristics of the installed autopilot.
- Control information: In order to support the different types of requests, each generated waypoint has some extra information associated to it that enables the FPM to tell which leg this waypoint belongs to, the iteration it was generated in, etc. This control data is used to resume waypoint generation at the right point when a change in the flight plan invalidates waypoints that have already been generated.

Figure 5.7 provides an overview of the internal classes that form part of the FPM. The *Controller* accesses the flight plan object model only when it needs to be updated and lets all

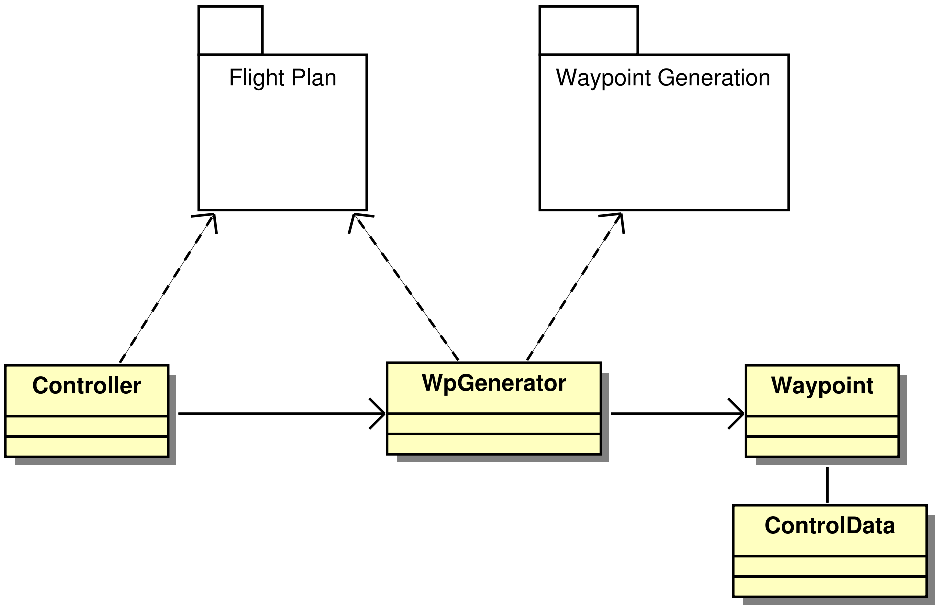


Figure 5.7: *FPM main classes.*

the waypoint generation process be handled by the *WpGenerator*. Structural representation of the flight plan is kept apart from waypoint generation classes following a separation of concerns principle. Finally, each generated waypoint is tagged with some control information so that it can be properly tracked when dynamic changes to the flight plan occur.

The *Controller* contains a main loop that, while in the *Auto* state, actively takes waypoints from the generation queue and sends them to the VAS (see Listing 5.1). If the FPM state changes to *Paused* or *StandBy*, it stops executing and waits until the *Resume* command is received. If we are switching to the *Paused* state, a notification is sent to the VAS, so that a holding pattern is flown while being in that state.

Listing 5.1: Controller main loop.

```
while (!done)
{
   switch(state)
   {
      case Auto:
         if (SentWpsCount < SENT_WINDOW)
         {
            Take waypoint from the generation queue;
            Send waypoint to the VAS;
            break;
         }
      case Paused:
         {
            Command VAS to perform holding pattern;
            Wait for Resume message;
            break;
         }
      case StandBy:
            Wait for Resume message;
   }
}
```

All requests sent to the FPM are managed in an event-driven fashion. Since some of them, as it would be the case for flight plan updates, may cause invalidation and re-generation of waypoints, thread synchronization mechanisms are used to keep both the generation and sent queues in a consistent state.

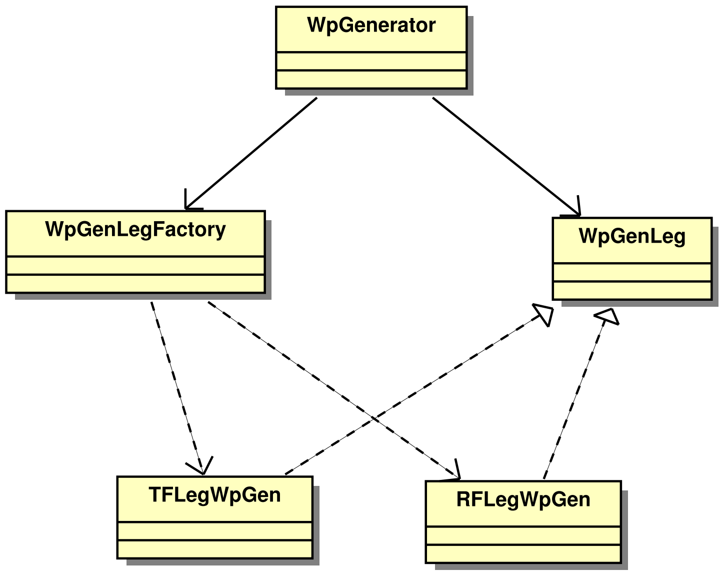


Figure 5.8: *A factory class is used to obtain leg specific waypoint generators.*

5.4 Waypoint Generation

This section provides details on how the waypoint generation process takes place. To this end, an iterative algorithm traverses the flight plan object model and computes waypoints for each of the legs encountered in the current path. A simplified view of the main loop is shown in Listing 5.2.

Listing 5.2: Generator main loop.

```
current_leg = Get first leg from the flight plan;
while (!done)
{
   Get generator object for current_leg;
   Compute waypoints for current_leg;
   Add waypoints to the generation queue;
   current_leg = Get next leg;
}
```

The first step in the loop body gets the object that will be used to generate waypoints for the current leg. Waypoints may be generated differently for each leg type, however, there is also the possibility to use different approaches for a single leg type depending on the VAS (or rather the underlying autopilot) capabilities. Therefore, the structural description of legs is separated from the generation algorithms used on them. There are several reasons for that: (1) First, the internal representation of the flight plan may be used in places where waypoint generation is not concerned. (2) By doing so, we also adhere to the single responsibility principle, so that changes to the generation process have no impact on the description classes. (3) Finally, this enables us to easily select one generation method between the potential alternatives.

As shown in Figure 5.8, a factory class (*WpGenLegFactory*) is used to obtain the waypoint generator (*WpGenLeg* derived classes) that suits the current leg. For instance, if a Track to a Fix is passed to the factory object, it will return an object able to generate waypoints for the Track to a Fix leg type. The actual type of the returned object will also depend on the capabilities of the autopilot. All generation classes derive from a common abstract class (*WpGenLeg*) and share the same interface.

Now the actual generation of waypoints can take place. As seen in Figure 5.9, some parameters are required to perform this computation:

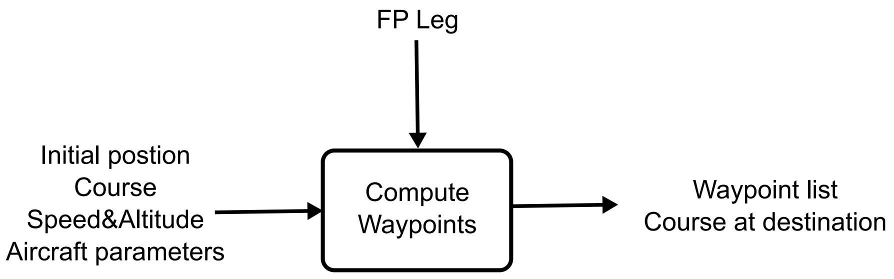


Figure 5.9: *Inputs and outputs to/from generator objects.*

- *Initial position* is the position of the aircraft at the beginning of *current_leg*.
- *Course* is the path (angle) the aircraft is following when reaching the initial position.
- *Speed* & *Altitude* are the estimated values the initial position is reached at.
- *Aircraft parameters* are the aircraft's bank angle and a correction factor to account for the transition time required to reach it.

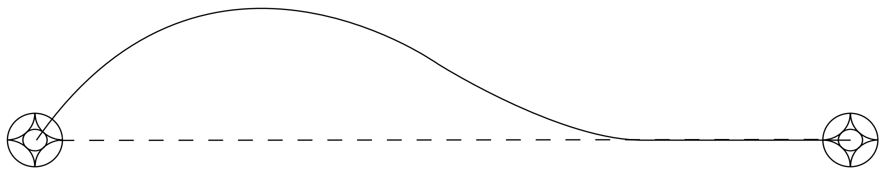
Although not yet considered in the current implementation of the FPM, other parameters, such as wind speed, should also be taken into account. As a result of the computation, a list of waypoints is obtained together with the predicted course at the last waypoint. The list of waypoints is added to the waypoint queue and the new course is used in the computation of the next leg.

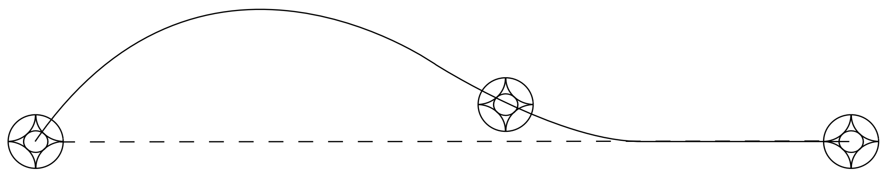
The described approach facilitates addition of new kinds of legs. In order to add a new leg, for instance a parametric one, we need to do the following steps:

- Implement a new description class that derives from *Leg* and overrides some of its methods. This class contains the parameters that are necessary to determine the trajectory represented by this leg.
- Implement one or more classes that derive from *WpGenLeg* and are able to generate the list of waypoints for the new leg type. These classes need to override some of the methods found in *WpGenLeg* so that the waypoint generator can transparently pass in the aircraft's state and receive the list of generated waypoints. The generation algorithm can take advantage of other leg types and create the list of waypoints of the current leg combining results from the others. In this way complex trajectories can be generated from basic legs.
- Modify *WpGenLegFactory* so that each time the new leg is found it is able to return the appropriate object for the waypoint generation process.

Our flight specification language and waypoint generation process emphasizes lateral navigation. The way in which vertical navigation takes places will depend on the underlying UAS autopilot. During the waypoint generation process, when multiple waypoints are generated from a given leg and the initial altitude differs from the altitude at the destination, altitude of intermediate waypoints will linearly be increased or decreased according to the lateral distance between each other.

While the generation process applies to all legs, there are some aspects that are particular to each kind of leg. These particular aspects are discussed in the following sections.


(a) *Fly-Over followed by Track to a Fix*


(b) *FO+TF with Fly-Over/Direct to a Fix capable system*

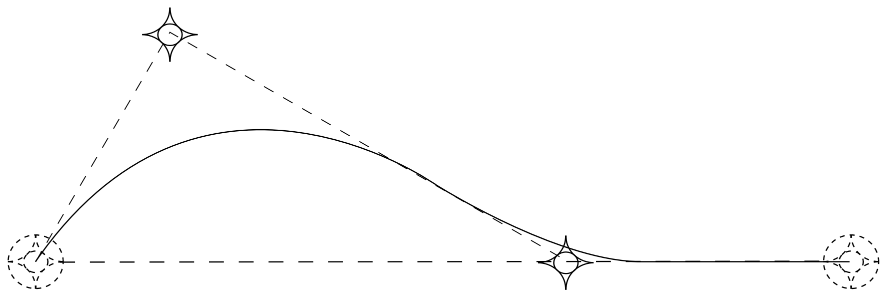

(c) *FO+TF with Fly-By/Track to a Fix capable system*

Figure 5.10: *Waypoint generation depending on autopilot capabilities.*

5.4.1 Basic Legs

The system currently supports five types of basic legs: Initial Fix, Track to a Fix, Direct to a Fix, Radius to a Fix and holding patterns. In some cases, waypoint generation is trivial, as an example, generating waypoints for an Initial Fix is accomplished by just adding the fix to the queue of generated waypoints. In other cases, waypoint generation is far more complex. Waypoint generation for a holding pattern takes into account that different entry procedures may need to be executed depending on the direction the aircraft comes from.

Another aspect that adds complexity to the generation process is that, while waypoint based navigation is a common denominator of the vast majority of UAS autopilots, it is unclear what their capabilities are with regard to their ability to stick to a given track or to perform both fly-by and fly-over waypoints. Figure 5.10 shows an example illustrating how these restrictions can be overcome with smart waypoint generation techniques that have into account the system capabilities. As a direct benefit from keeping structural leg data and waypoint computation separated in different classes, we are able to pick the generation algorithm that best suits each situation.

In Figure 5.10a, we see what should be the trajectory for performing a Track to a Fix having a fly-over waypoint as the initial aircraft position. Once the starting waypoint has been over-flown, the aircraft turns right in order to intercept the track. Figures 5.10b and 5.10c respectively show how this same trajectory can be obtained with an autopilot system that only supports Direct to a Fix navigation and with one that only implements fly-byes. In both cases, additional waypoints are strategically added and others removed so that the intended trajectory is achieved. In Figure 5.10b, an extra waypoint is added so that the aircraft is forced to turn twice in order to reach the destination. In Figure 5.10b, the two fly-over waypoints at the beginning and the end of the trajectory are replaced by two fly-by ones at different positions. The details on how to carry out the actual computations for these and other cases can be found in (Trillo, 2009).

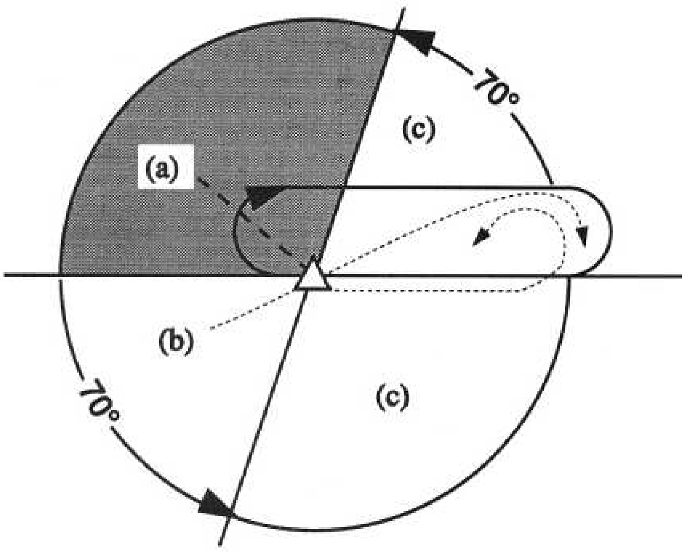


Figure 5.11: *Standard holding entry procedures.*

The replacement of the first waypoint in Figure 5.10b can actually be seen as taking the destination of the previous leg and move it further along the aircraft's course. Therefore, the final waypoint of a given leg may actually depend on what happens in the next leg. To handle this situation, the generation process for a given leg is performed in two steps:

1. Generate waypoints for the current leg regardless of what happens next.
2. Every time a new leg is added to the waypoint list, check if the destination of the previous one needs to be corrected.

When computing waypoints for an RF leg, the system will approximate the turning maneuver with a sequence of waypoints intersecting the desired path on those systems that support fly-over. If the system makes use of fly-by waypoints, generated waypoints will be slightly displaced from the desired trajectory. The distance between consecutive waypoints is directly proportional to the aircraft's turning radius which, in turn, depends on the aircraft's speed and its bank angle, the latter value being a constant that characterizes the aircraft.

Holding patterns are generated by concatenation of TF and RF legs and, in that respect, are very similar to the way parametric legs are generated. Another aspect that differentiates HF legs from the rest of basic legs is that they require an entry procedure that depends on the angle the aircraft comes from. Figure 5.11 shows, in dashed lines, the three different entry procedures. Different regions around the HF leg indicate the different directions the aircraft may come from. Below each entry procedure is explained:

- A parallel entry is executed when the holding fix is approached from sector (a). After reaching the holding fix, the aircraft parallels the inbound course, then turns back and returns to the holding fix to continue the hold from there.
- A teardrop procedure applies when coming from sector sector (b). In this case the aircraft flies to the holding fix, turns into the protected area and then turns in the direction of the holding pattern to intercept the inbound holding course.
- In a direct entry procedure, which is executed when approaching the holding fix from anywhere in sector (c), the aircraft flies directly to the holding fix and immediately turns to follow the holding pattern.

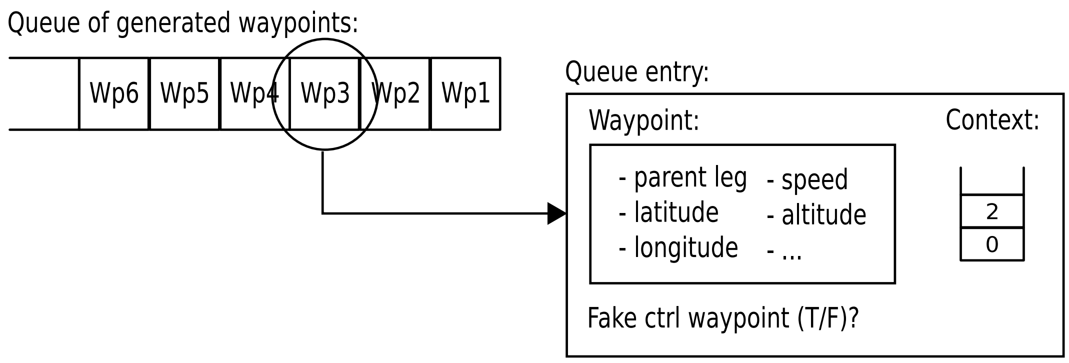


Figure 5.12: *Contents of waypoint queue entries.*

5.4.2 Iterative Legs

An iterative leg is a control structure used to specify that certain parts of the flight plan should be repeatedly executed. By itself it does not determine any kind of trajectory. It just groups together a number of legs that may be executed several times.

Dealing with iterative legs implies that the waypoint queues will contain waypoints coming from different instantiations of a single leg. Moreover, iterative legs can be nested so that two given waypoints may have been created for the same iteration of an inner leg but different iterations of an outer one and vice versa. To be able to track what iterations a given waypoint, and its corresponding leg instantiation, belong to, all waypoints are tagged with context data. As seen on Figure 5.12, each enqueued waypoint contains all data directly related to the waypoint, such as parent leg, latitude, longitude, etc. plus a stack of integers. Each time a new iterative leg is entered an integer value of zero is added on top of the stack. This value is incremented at each iteration and popped out when no more iterations are left. In the example shown in Figure 5.12, *Wp3* belongs to a leg nested within two iterative legs and has been generated during the first iteration of the inner one and the third iteration of the outer one.

Keeping track of the context information is crucial due to the iterative nature of the generation process. As seen on Listing 5.2, flight plan legs are taken one at a time and it does not suffice to say what leg should waypoints be generated for. It is mandatory to know what exact iterations of enclosing iterative legs have already been processed and which are the current ones, otherwise, we would not be able to tell when waypoint generation for a given iterative leg has finished.

Every time an iterative leg is encountered the waypoint generator checks whether we are starting fresh or this is an additional iteration of an ongoing iterative leg. It also checks if the upper bound has been reached and what is the outcome of a possible associated condition. The iterative leg does not generate any waypoints by itself. If it is determined that a new iteration needs to be generated the first leg of its body will be returned when the main generation loop asks for the next leg. To leave a trace of the presence of the iterative leg in the waypoint queue, a fake waypoint is added. Therefore, also shown in Figure 5.12, all queue entries are marked as being fake or not fake. Fake waypoints are not sent to the autopilot.

5.4.3 Intersection Legs

Like iterative legs, intersection legs are also control structures that do not determine a trajectory. The purpose of intersection legs is to enable the possibility to choose between different alternative paths. When an intersection leg is found only waypoints belonging to the selected path are generated. Should the value of its governing condition change, all waypoints found in the FPM's queues from that point onwards will be discarded and new waypoints will be generated starting at the intersection.

Each time an intersection leg is found, a fake waypoint is added to the waypoint queue. In this way, when its condition changes it can easily be found. Once invalid waypoints have been discarded, the *current_leg* variable that appears in Listing 5.2 is set to the intersection leg. We also take advantage of the control information associated to each waypoint to be able to properly restart the generation process.

If the intersection leg condition has not been given a value and there is no default leg, waypoint generation is interrupted. This may result in the VAS running out of waypoints. If this situation happens, once all waypoints have been consumed, the FPM will command the VAS to perform a holding pattern until new waypoints become available.

5.4.4 Parametric Legs

Parametric legs are used to generate complex paths, or paths that would otherwise require a large number of legs, from a reduced number of input parameters. While the available number of parametric legs is expected to steadily grow so that a wide range of mission can be supported, an eight pattern leg and a scan leg that covers a rectangular area have already been implemented. Both leg types are put to use in Chapter 8 to illustrate how the FPM and the MMa work together to carry out a hotspot detection mission.

When generating waypoints for an eight pattern, TF legs and RF legs are combined to respectively perform the straight and turning parts of the maneuver. The same approach is followed for the rectangular area scanning pattern. In this case, prior to waypoint generation, the area of interest that will be covered needs to be computed. Then, using TF and RF legs as the construction elements, all passes over the area are generated with the separation distance indicated in the definition of the leg. To improve its flexibility an additional parameter can be used to set an arbitrary waypoint within the area of interest as the starting point of the scan procedure. In the example shown in Chapter 8 we take advantage of this feature to be able to stop its execution, do something else, and then come back and continue where the scanning was interrupted.

Making use of other basic legs for generating waypoints has the added benefit that all special processing related to overcoming autopilot limitations is transparently done.

5.5 Dynamic Flight Management

This section summarizes the most relevant details related to the implementation of flight management features that enable more dynamic control over the flight path. Such features include skipping legs, managing updates and executing emergency procedures.

5.5.1 Skipping Legs

The FPM supports two ways to skip parts of the flight plan: a *Skip* leg command to ignore the rest of the current leg and a *Goto* command that skips any number of legs and directly jumps to the indicated one. The former being just a special case of the latter, both commands are implemented in the same way.

While, at first glance, it may seem that discarding all intermediate waypoints suffices to implement this feature, in reality we cannot guarantee that waypoints following the destination of the jump are still valid. A clear example of that is when jumping directly into a holding pattern, whose entry procedure depends on the direction the aircraft comes from.

To implement the *Skip* and *Goto* commands we first locate the destination leg in the FPM's queues. Once located we extract the context data that is needed for properly setting up the restart point. A temporary DF leg is created with the *Goto* destination as its next one. This new leg, together with the context information, is used to restart waypoint generation once all the pending waypoints have been removed from the queues and from the VAS.

5.5.2 Managing Flight Plan Updates

The hardest part of a flight plan update is to actually process the update message and make the internal representation of the flight plan reflect those changes. In the best of cases, these updates will only modify some attributes of one or more legs without really affecting the flight plan structure. This kind of updates are already supported for all available leg types and already provide a powerful means to adapt the UAS path to the mission needs. Consider, for instance, a situation where the size of an area under inspection changes, or where the position of a particular point of interest is unknown prior to the execution of the mission.

Support for more aggressive updates that enable legs to be dynamically added and removed from the flight plan is also planned and the form these kind of updates will take has already been presented in Chapter 4. The main difficulty for supporting them lies in the fact that the flight plan must be checked for consistency when changes are applied.

Once the structural representation of the flight plan reflects submitted changes, the way to proceed with regard to the waypoint generation process is not different from what happens when the value of a given condition is modified. If no waypoints have been generated yet for the affected legs, or if they have already been flown, waypoint generation can continue without requiring any special action. If there are pending waypoints in the FPM's queues that belong to some of the legs involved in the update, these waypoints need to be discarded, with the corresponding notification to the VAS if they have already been sent. The generation process needs then to restart at the point where the oldest discarded waypoint was found. Therefore, the process for restarting waypoint generation consists in first, locating the updated leg whose waypoints are closest to the head of the waypoint queue, and then, having identified this leg, changing *current_leg* in the generation loop. Context data, that can be extracted from the waypoints in the queue, also needs to be taken into account.

5.5.3 Loosing and Regaining Navigation Control

The method for the FPM to resume navigation control when returning from the *StandBy* state will depend on the circumstances that caused control to be taken away from the service. If navigation control was taken away for performing an emergency maneuver to avoid collision with an obstacle, it may not be a good idea to return to the position where the obstacle was found. The situation is completely different if the obstacle was a moving object or an onground pilot just momentarily switched to manual control to observe an area not covered by the flight plan. The FPM will not know what really happened during the period navigation was not under its control. For this reason, it will implement three strategies for continuing its operation. The *Resume* command will contain a parameter to select which one should take place. The three possibilities consist in:

- Fly back to the last flown waypoint before the FPM was interrupted.
- Fly back to the position where the aircraft was at the moment the interruption occurred.
- Fly to the waypoint the aircraft was heading to when the interruption occurred.

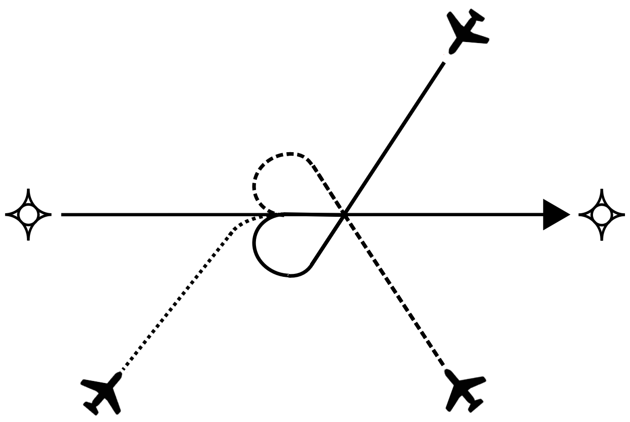


Figure 5.13: *Maneuvers for resuming flight at a given point.*

Waypoint generation for a given leg depends on the previous one. We need to know what speed, altitude and course the leg is reached at. In other words, if these parameters vary, the actual waypoints resulting from the generation process for a given leg may also vary. This means that, when the FPM tells the autopilot to fly to a given waypoint, the speed, altitude and course conditions that where in place when the waypoint was generated must be reproduced. To be able to do so, all these parameters are stored within each generated waypoint. An entry maneuver, that may consist in a smooth turn or a more complex holding-like entry procedure, will automatically be generated to ensure that the aircraft reaches the waypoint with the appropriate course. Figure 5.13 displays how a particular point should be reached depending on the initial aircraft position.

If none of the previous methods is satisfactory, a *Goto* command can always be issued in order to resume execution at a given forthcoming leg.

5.5.4 Execution of Emergency Plans

The FPM has been designed so that there is no difference between flying the main flight plan or an emergency alternative. Legs are treated the same way and the waypoint generation process is the same. The only time when some special actions are required is during the transition phase.

The first step consists in determining what emergency plan needs to be executed. Although the FPM stores, provides information, and manages emergency flight plans, it is not responsible for deciding what emergency plan needs to be executed and when it needs to be executed. This responsibility belongs to the Contingency Manager, that will trigger an event whose parameters will identify the selected emergency plan.

Figure 5.14 shows how the CM obtains the flight time duration of available emergency plans for the current flight phase. When the CM detects a hazardous situation, the MMa gets informed. The MMa may then reconfigure payload operation, for example, in order to maximize battery life. After that, the CM selects the emergency plan and this decision is notified to the FPM.

Once the emergency alternative is known, the entry point to the emergency alternative is computed. This computation is based on distance and the entry point which is closest to the current position will be selected.

At this point, the current flight plan can be replaced by the emergency one and its execution can start. This implies discarding all waypoints that do not belong to the emergency plan in all queues and in the VAS and, afterwards, generate a DF leg that takes the aircraft to the selected entry point and continue from there. This last step can be performed making use of the previously discussed *Goto* command.

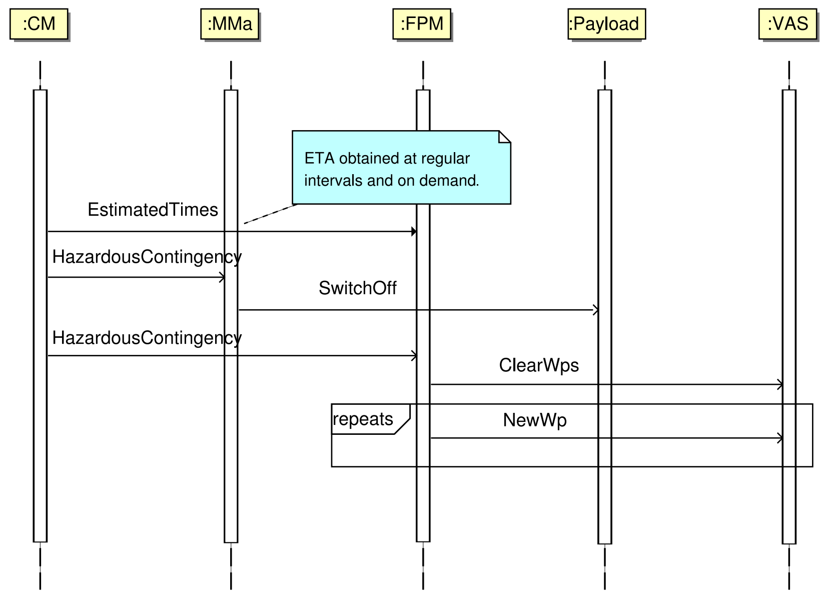


Figure 5.14: *FPM role in managing contingencies.*

5.6 Conclusion

This chapter has provided the main design and implementation details of the Flight Plan Manager. The main goal of the FPM is to extend the capabilities of UAS autopilots and enable the execution of flight plans expressed in our specification language. It also implements the logic that enables the flight plan to be dynamically updated and provides operations to control the UAS flight.

FPM capabilities can be compared to the flight planning functionality of a Flight Management System (see Section 2.2.2). The FPM also plays a role in navigation, since it generates the waypoints the aircraft must fly to, but it is responsibility of the autopilot to determine the current UAS position and implement the guidance and lower level control loops. Other functions that a FMS may implement include trajectory prediction and performance. It would be interesting to see how far we can go trying to implement similar capabilities, having into account that the FPM is intended to operate on top of a commercial autopilot, but this problem is not tackled in this thesis.

Our current implementation of the FPM should be seen as a proof-of-concept prototype. One of the aspects that needs to be addressed in a production version is guaranteeing flyability, i.e., ensure that the UAS platform is able to fly the intended maneuvers considering speed, altitude, turning radius, climb rates, etc. Another element that needs to be addressed is the inclusion of wind effects in the waypoint generation process.

6 Flight Plan Experimental Results

This chapter gives the results obtained in the application of the flight plan specification and execution methods to a hypothetical Radio Navigation Aids (navaids) flight inspection mission. The use of UAS for this kind of application has been proposed in (Ramirez *et al.* , 2009). The experimental results have been obtained in the simulation environment described in Section 6.3.

6.1 Navaids Flight Inspection Mission

The current Air Transportation System relies on the use of Radio Navigation Aids to provide the capability to fly, in a safe manner, with unfavorable visibility conditions. These navaids are subject to inspections that verify the adequacy of the radio-frequency emission to the standard. While some of these inspections could be conducted on ground, for some of the inspected magnitudes there is a set of measurements that shall be obtained in air by means of a flight inspection. Flight inspection has been conducted for many decades in many countries. This experience is reflected in different standards (ICAO, 2000; FAA, 2005) which compile magnitudes, flight procedures, criteria for accepting the inspection etc.

A requirement for the flight inspection is to minimize the impact on the rest of airspace users. This is reflected in the interruption of some procedures if another aircraft enters into the inspection area. Nowadays, flight inspection is being conducted with general purpose aircrafts conveniently instrumented and with skilled personnel that perform the flight inspection on board. The ability to interrupt and resume the inspection procedures is provided by the cabin crew.

The current approach satisfies the aviation authorities and the Air Navigation Service Providers (ANSPs) technical and operational requirements but it is expensive. The flight

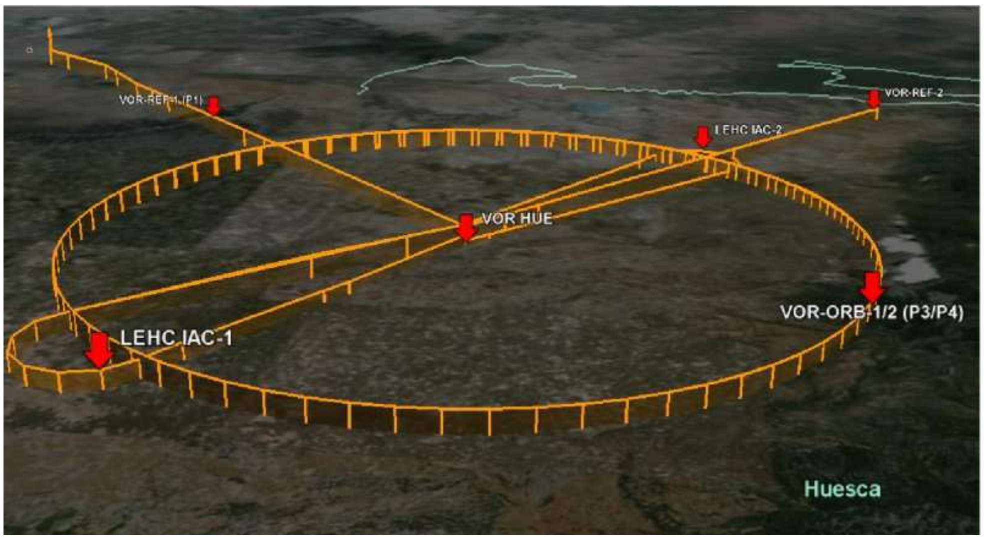


Figure 6.1: *VOR navaid at Huesca, Spain.*

inspection community has proposed many improvements to reduce costs (Qvist, 2006; Wede, 2006), e.g., by reducing the flight time of the flight inspector sitting him on ground and providing the data through telecoms. J. Ramirez et al. propose going further and make use of UAS (Ramirez *et al.* , 2009).

By removing the flight crew, the source of agility in the aircraft is also removed and this introduces a new requirement to the flight inspection platform: The operational agility of the UAS flight inspection platform shall be equivalent to the conventional flight inspection platform.

During the flight inspection of a navaid, different measurements of the signal must be taken. These measurements shall be performed following procedures as detailed in ICAO doc 8071 (ICAO, 2000). The final definition of these procedures depends both on the generic procedures for the navaid type (VOR, DME, ILS...) and on the operations supported by the system inspected (e.g., SID or STAR of an airport).

The flight inspection is performed coordinately with the ANSP who provides navigation services with the inspected navaid. For safety reasons, during flight inspections on small aerodromes, the air traffic is restricted except for the flight inspection aircraft. In big airports, where restricting traffic is extremely expensive, the flight inspection aircrafts are inserted in normal air traffic. In both cases, the ATC of the region/aerodrome where the flight inspection takes place is aware of the special operations required for the flight inspections and, in some cases (e.g., in crowded airports), the ATC is replaced temporarily by an ATC specially trained for flight inspection. The flight inspection procedures detailed in the standards require the acquisition of some physical magnitudes in specific trajectories (e.g., an orbit around the navaid, a straight line over the facility...). The overall set of trajectories and measurements for a specific VOR navaid is shown in Figure 6.1. This figure shows the complexity of the simulated flight inspection mission whose results are given in the following sections. This mission corresponds to a real facility located in Huesca, Spain.

Each of these trajectories needs an entire set of data without errors for evaluating its adequacy to the standards. The completeness and correctness of these sets could be altered by measurement errors or by modifications of the trajectory imposed by ATC for solving air traffic conflicts.

There are different sources for the errors observed in the flight inspection. These errors could be originated by the navaid itself, in which case the facility should be turned off, or by external causes (e.g., an aircraft crossing the runway and intercepting the ILS signal), in which case the measurement should be repeated to verify its correctness. For economy reasons this repetition is performed only over the segment affected by the suspected data.

The envisaged scenario for UAS integration contemplates the division of the system into two segments: the aerial vehicle and the ground station. The ground station, dedicated to the control of the aircraft, is different from the ATC positions dedicated to air traffic. The ground segment shall stay in contact with ATC and issue the commands necessary for making UAS honor the ATC orders.

During a flight inspection ATCs are aware of the kind of mission being conducted. Specially trained ATCs are designated for the purpose and will usually minimize the impact of ATC coordination in the flight inspection operation. Nevertheless the combination of flight inspection and airspace integration could oblige ATC to impose some alterations on trajectories for conflict avoidance purposes. The usual means are:

- Speed variation
- Altitude variation
- Direction variation
- Airspace temporal use denial

Usually ATC use speed and altitude variations for conflict resolution. The flight inspection measurements remain valid with speed variations but the change in the altitude invalidates the flight inspection data (ICAO, 2000; FAA, 2007). In crowded scenarios ATC could use direction variation (also known as ATC vectors). In this case the flight inspection measurements are no longer valid, as they are captured outside of the required trajectories. The trajectories shall be flown again in order to finish the inspection mission. The denial of airspace use is extremely rare. In this case, the mission is aborted and the flight inspection remains unfinished until the airspace is reopened.

The previous ATC orders can be analyzed by its mission implications and the possibility of interrupting the mission:

- Mission termination
- Mission suspension
- Repetition of one or more legs

Mission termination could be motivated by several causes. The inspected navaid being clearly out of service is the most evident, but can also respond to ATM or civil/military interoperability needs. If the mission is aborted, a return to home has to be performed.

Mission suspension may be required when another aircraft is allowed to enter the zone where the inspection is conducted, creating a conflict with the flight inspection aircraft. In this case the flight inspection is temporarily interrupted to avoid incidents in mid air until the intruder aircraft has finished its operation. Afterwards the flight inspection operation is resumed.

Repetition of one or more legs obeys to flight inspection needs. If a measurement is suspected to be erroneous the flight is repeated to disambiguate the origin of the error.

This alteration of the mission motivates dynamic modifications on the preflight established mission duration that shall be controlled in order to not surpass the autonomy of the aircraft. This unexpected nature of the different interruptions affecting flight inspection missions adds an additional interruption to be considered: the lack of fuel. Lack of fuel shall be managed as a mission termination.

Next section describes the procedures that need to be flown during the flight inspection of the aforementioned VOR navaid located in Huesca, Spain. This procedures are translated to the proposed flight plan specification language and control constructs are added to cope with the agility requirements.

6.2 Inspection Procedures

In the previous section the main aspects of the flight inspection application have been introduced. Now, the specific procedures for the periodic flight inspection of the VOR/DME located in Huesca, Spain are provided.

In order to fulfill the mission requirements the system must be able to:

- Fly all procedures as well as or better than conventional flight inspection systems.
- Repeat any procedure or part of it if the results are not satisfactory.
- Interrupt the pre-established flight plan according to previously exposed interruption causes.
- Continue the flight inspection procedures where they were interrupted according to efficiency issues.

6.2.1 Procedures for a periodic flight inspection of VOR/DME navaid at Huesca, Spain

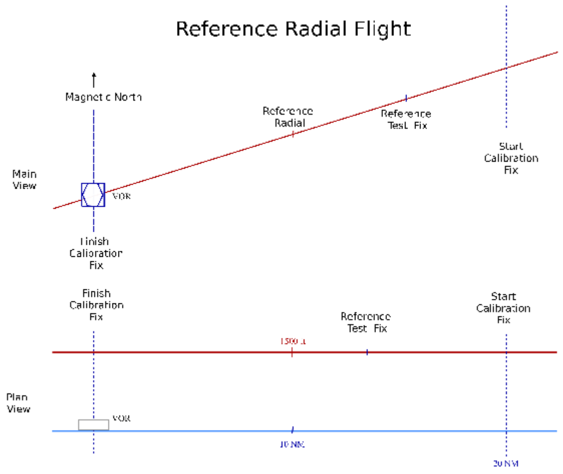
There are seven procedures that have to be flown in order to perform a periodic flight inspection of a VOR/DME navaid (ICAO, 2000; FAA, 2007; NATO, 2000). These procedures can be described in terms of three of the basic legs developed in Chapter 4: Direct to a Fix (DF), Track to a Fix (TF) and Radius to a Fix (RF). Table 6.1 lists all periodic flight inspection procedures with indications of the legs that are going to be used.

Reference radial and orbital procedures have to be flown at the same altitude. Due to the terrain orography in the vicinity of Huesca airport the minimum height to fly these procedures safely is 1410 meters.

Table 6.1: *Procedures for a periodic flight inspection*

Procedure number	Procedure type	Type of Leg used
VOR-REF-1	Reference Radial Flight	TF
VOR-REF-2	Reference Radial Flight	TF
VOR-ORB-1	Orbital Flight	RF
VOR-ORB-2	Orbital Flight	RF
VOR-RAD	Radial Flight	TF
VOR-APP	Approaches	TF, RF
VOR-SIDSTAR	SID, STARS	TF, DF

Procedure	Start	Finish	AGL Height
Horitzontal Flight to Aid	20 NM	Aid	1500 ft or secure minimum height



```
<leg id="VOR–REF–1–A" xsi:type="fp:TFLeg">
  <dest>
    <coordinates>
      41.818148 −0.659859
    </coordinates>
    <fly−over>true</fly−over>
    <altitude>2000</altitude>
    <speed>200</speed>
  </dest>
  <next>VOR–REF–1–B</next>
</leg>
<leg id="VOR–REF–1–B" xsi:type="fp:TFLeg">
  <dest>
    <coordinates>
      42.073333 −0.318888
    </coordinates>
    <fly−over>true</fly−over>
    <altitude>2000</altitude>
    <speed>200</speed>
  </dest>
  <next>VOR–VOID–1–A</next>
</leg>
```

Figure 6.2: *Reference Radial Flight.*

The procedure order is determined by two factors. *VOR-REF-1* and *VOR-REF-2* have to be flown first because they test vital parameters. The other procedures are ordered according to efficiency. The lesser flight time the better.

6.2.1.1 Reference Radial Flight (VOR-REF-1 and VOR-REF-2)

This procedure consists in flying a VOR navaid radial at constant altitude. The main objective of this procedure is comparing vital parameters (such as magnetic deviation) with the record of previous inspections. It is flown twice because the VOR navaid has two transmitters due to redundancy aspects. Both of them transmit at the same band, hence, they have to be tested separately.

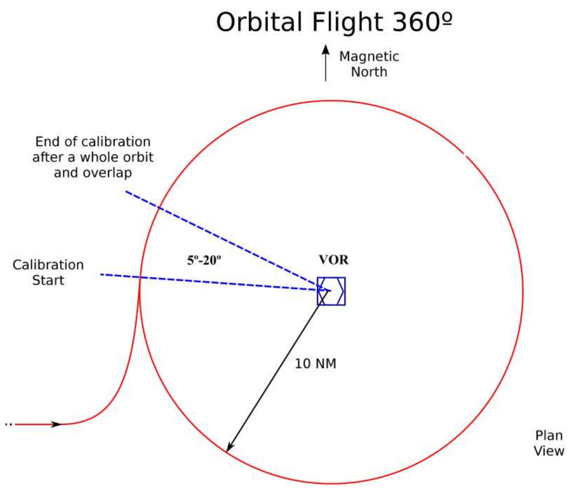
Figure 6.2 shows a table with the parameters that characterize the Radial Flight procedure. Also in the figure, we can graphically see how this procedure is mainly defined by two waypoints named *Start Calibration Fix* and *Finish Calibration Fix*. In order to specify this procedure two Track to a Fix legs are used, one per each waypoint. The first TF leg places the UAS at the beginning of the procedure. Then, the second one makes the UAS execute the procedure. The encoding of these legs using our flight plan specification language is also shown in Figure 6.2.

6.2.1.2 Orbital Flight 360 Degrees (VOR-ORB-1 and VOR-ORB-2)

This procedure is an orbital flight with constant radius. The center of the orbit is the navaid position. Its main objective is to determine if the signal coverage is between the established limits. Other parameters are also tested. For the same reasons as in the Reference Radial Flight, this procedure also needs to be flown twice.

The Orbital procedure is characterized by the parameters shown in Figure 6.3. This procedure is defined by three waypoints: The *Calibration Start*, the *End of Calibration* after a whole orbit, plus

Procedure	Start	Finish	Center	AGL Height
Orbital Flight	Anywhere in the orbit	Overlapping area between 5-20 degrees from the initial point	Aid	Same as Reference Radial



```
<leg id="VOR-VOID-2-A" xsi:type="fp:DFLeg">
  <dest>
    <coordinates>
      42.217328 -0.431756
    </coordinates>
  </dest>
  <next>VOR-ORB-1-A</next>
</leg>
<leg id="VOR-ORB-1-A" xsi:type="fp:RFLeg">
  <dest>
    <coordinates>
      42.157184 -0.125393
    </coordinates>
  </dest>
  <next>VOR-ORB-1-B</next>
  <center>42.0733331 -0.318888</center>
  <direction>Right</direction>
</leg>
<leg id="VOR-ORB-1-B" xsi:type="fp:RFLeg">
  <dest>
    <coordinates>
      41.927604 -0.210376
    </coordinates>
  </dest>
  <next>VOR-ORB-1-C</next>
  <center>42.0733331 -0.318888</center>
  <direction>Right</direction>
</leg>
<leg id="VOR-ORB-1-C" xsi:type="fp:RFLeg">
  ...
</leg>
<leg id="VOR-ORB-1-D" xsi:type="fp:RFLeg">
  ...
</leg>
```

Figure 6.3: *Reference Orbital Flight.*

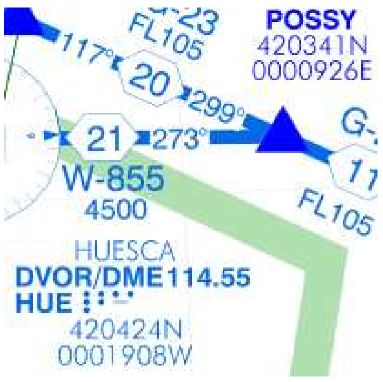
a 5° to 20° overlap, and the VOR as center of the orbit. To specify this procedure the required flight path is broken down into several Radius to a Fix legs. By doing so, we limit the extend that needs to be flown more than once in case the procedure cannot be completed in a single pass due to interruptions. An intermediate DF leg connects this procedure with the previous one.

6.2.1.3 Radial Flights (VOR-RAD)

In order to ensure the correct reception of VOR signal, all the VOR radials that are used to define airways shall be tested by flying them 100 ft below the specified altitude (terminal radial) or at the minimum secure altitude (en-route radial).

Figure 6.4 shows an airway (W-855) based on a terminal radial. This procedure is similar to the Reference Radial Flight procedure (*VOR-REF-1*, *VOR-REF-2*). The procedure is defined by two waypoints, one for the start point and one for the end point. First, a DF places the UAS at the beginning of the procedure. Then, the procedure gets executed using a TF.

Procedure	Start	Finish	Height
Radial Flight (terminal radial)	Published maximum range	Aid	Minimum secure altitude
Radial Flight (en-route radial)	As published at AIP	As published at AIP	100 ft below published altitude



```
<leg id="VOR-VOID-3-B" xsi:type="fp:TFLeg">
  <dest>
    <coordinates>
      42.061388 -0.157222
    </coordinates>
    <fly-over>true</fly-over>
    <altitude>1410</altitude>
  </dest>
  <next>VOR-RAD-1-A</next>
</leg>
<leg id="VOR-RAD-1-A" xsi:type="fp:TFLeg">
  <dest>
    <coordinates>
      42.073333 -0.318888
    </coordinates>
    <fly-over>true</fly-over>
  </dest>
  <next>VOR-VOID-4-A</next>
</leg>
```

Figure 6.4: *Radial flight example.*

6.2.1.4 Approaches (VOR-APP)

An Instrument Approach Procedure (IAP) is a type of air navigation that allows pilots to land an aircraft in reduced visibility or to reach visual conditions permitting a visual landing. In order to ensure the correct reception of VOR signal in these procedures, periodic flight inspection includes the flight of all approach procedures based in the inspected navaid.

All operations that conform an approach (see Figure 6.5) can be specified combining Track to a Fix and Radius to a Fix leg types. Encoding of these procedures is similar to other examples provided.

6.2.1.5 Standard Instrumental Departure (VOR-SIDSTAR)

Standard Instrument Departure (SID) routes, also known as Departure Procedures (DP) are published flight procedures followed by aircraft on an IFR flight plan immediately after taking off from an airport.

Figure 6.6 shows the specification for an Standard Instrumental Departure procedure. As it can be seen, it has been specified combining Track to a Fix and DF to a Fix leg types.

A Standard Terminal Arrival Route (STAR) covers the phase of a flight that lies between the top of descent from en-route flight and the final approach to a runway for landing. Such kind of procedure could be specified in similar terms.

6.2.2 Flight plan structure

In the previous section different procedures for a flight inspection were presented and specified. Since the inspection requirements demand supporting interruptions and restarts, some additional

Procedure	**Start**	**Finish**	**Height**
Approach	As published at AIP	As published at AIP	As published at AIP

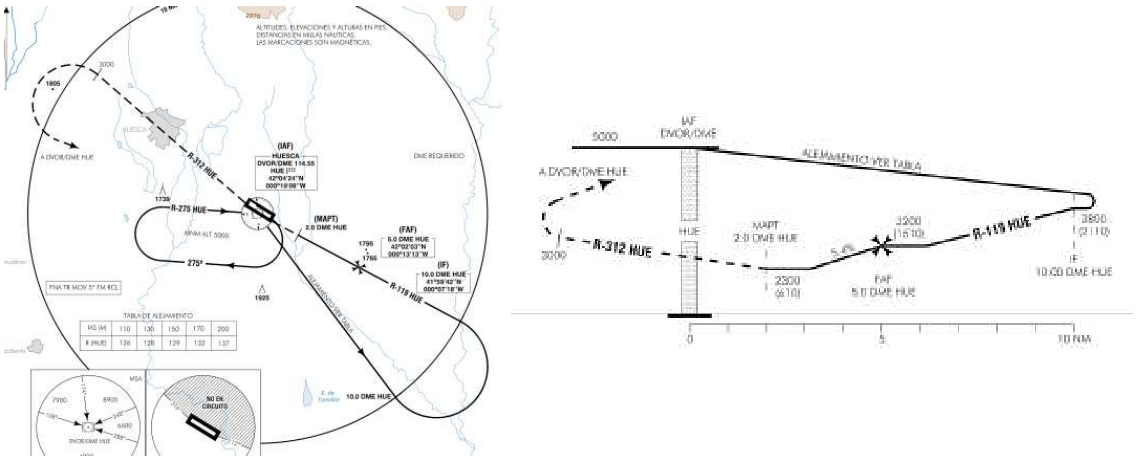
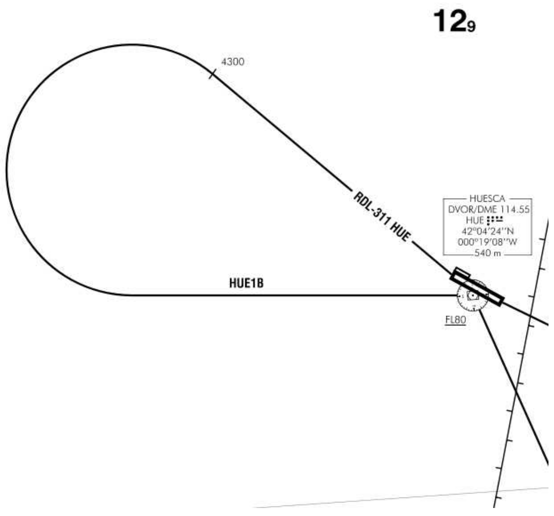


Figure 6.5: *Approach example.*

Procedure	Start	Finish	Height
SID	As published at AIP	As published at AIP	As published at AIP



```
<leg id="VOR–SID–1–A" xsi:type="fp:TFLeg">
  <dest>
    <coordinates>
      42.151350 −0.439552
    </coordinates>
    <altitude>1310</altitude>
    <speed>160</speed>
  </dest>
  <next>VOR–SID–1–B</next>
</leg>
<leg id="VOR–SID–1–B" xsi:type="fp:DFLeg">
  <dest>
    <coordinates>
      42.071417 −0.482818
    </coordinates>
    <altitude>1410</altitude>
    <speed>200</speed>
  </dest>
  <next>VOR–SID–1–C</next>
</leg>
<leg id="VOR–SID–1–C" xsi:type="fp:TFLeg">
  <dest>
    <coordinates>
      42.073333 −0.318888
    </coordinates>
  </dest>
</leg>
```

Figure 6.6: *SID example.*

legs need to be added to the flight plan. In particular, iterative legs are added to group together repeatable blocks of inspection legs. In addition to that, intersection legs are used to provide alternative paths that lead to the execution of holding procedures. Figure 6.7 shows the resulting organization of the flight plan.

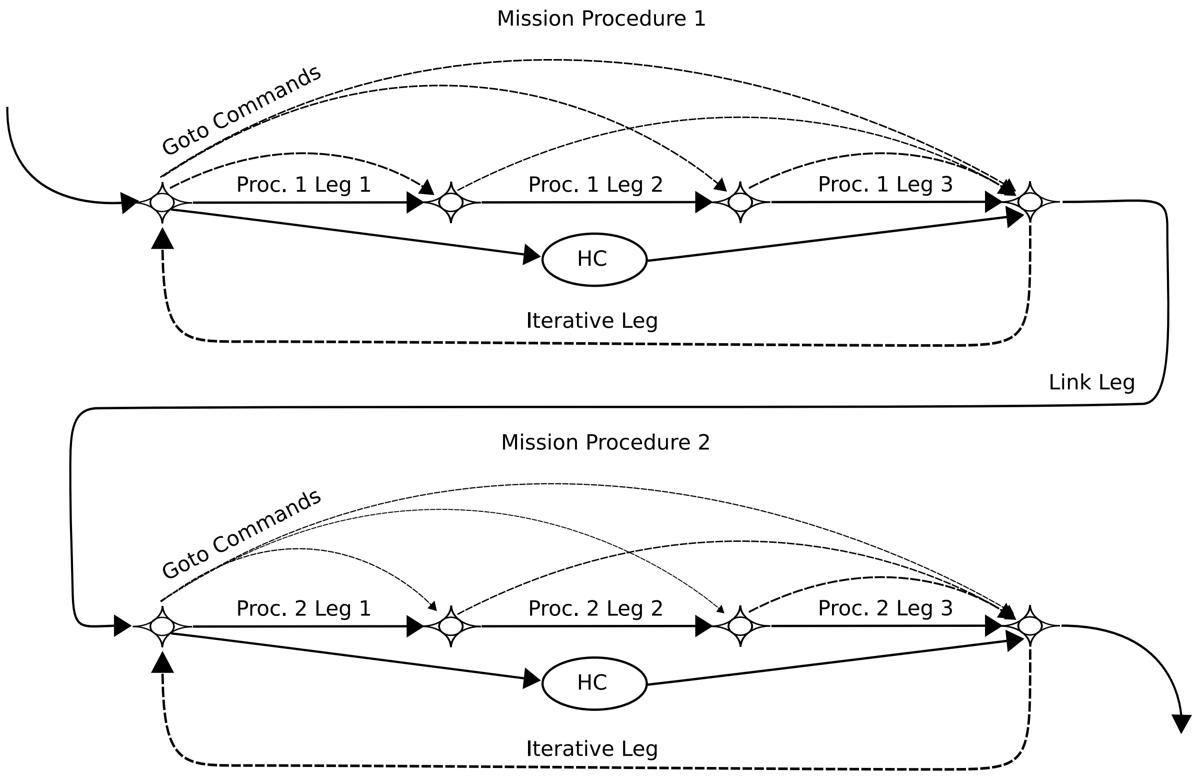


Figure 6.7: *Flight plan organization.*

Each iterative leg contains one main path, with an inspection procedure divided into consecutive legs, and an alternative path, with a Holding to a Condition leg. An intersection leg at the beginning of each iterative leg allows selection between the main path, that executes the inspection procedure, and the holding pattern. In this way the operator can switch from one to the other.

To minimize the extend of the flight path that needs to be repeated when an interruption occurs, the flight plan designer divides each procedure into smaller legs. Then, using the *Goto* command, the operator has the ability to directly jump to the desired leg and proceed from there.

6.3 Simulation Environment

The simulation environment used for testing our flight plan specification language and its execution engine is depicted in Figure 6.8. Boxes on top of the Network bar represent embarked services. To perform the simulation only the Virtual Autopilot System (VAS) and the Flight Plan Manager (FPM) are required. The aircraft behavior is simulated using the FlightGear flight simulator (Olson, 2010), an open-source project licensed under the GNU General Public License.

Boxes below the network bar belong to the ground segment, these include a Ground Control service to control the UAS operations and a Flight Tracking System that displays the UAS trajectory in real time using Google Earth®.

For the purpose of the simulation, a Beechcraft B1900D has been used. This is one of the aircrafts available in FlightGear's models database and is commonly used in inspection operations. Figure 6.9 shows some of the simulation parameters. Experimentally it has been determined that the bank angle used by this model for turning is 30 degrees. A roll factor of 14 seconds accounts for the time it takes to reach the bank angle. The simulation has been run without

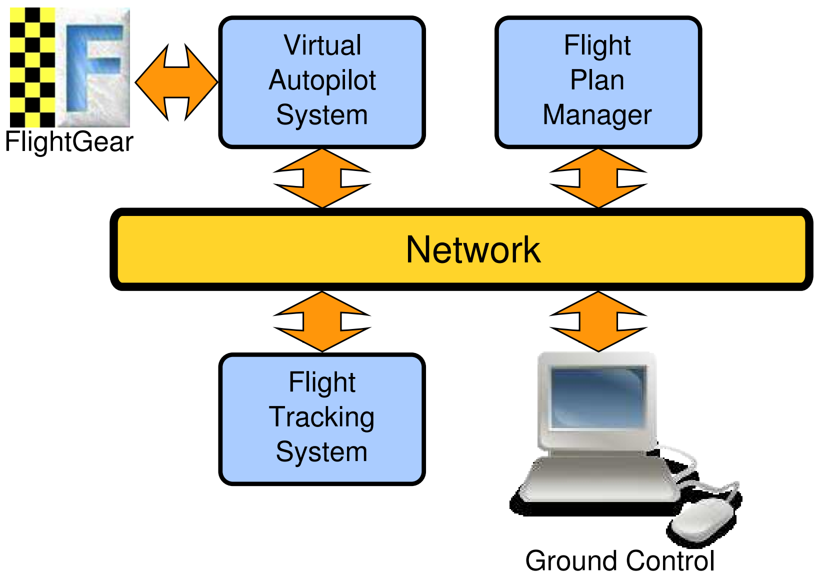


Figure 6.8: *Simulation environment.*

Aircraft	Beechcraft B1900D
Cruise speed	230 kt (425 km/hr)
Bank angle	30 °
Roll factor	14 s
Wind	no wind
Fuel	unlimited
Mission duration	1 h 30 m aprox.

Figure 6.9: *Aircraft and simulation parameters.*

wind and with unlimited fuel, so that the simulation time is not constrained by the aircraft's autonomy.

As discussed in Chapter 3, the VAS provides waypoint navigation capabilities and a number of telemetry flows regarding the UAS position, attitude, autopilot status, etc. A critical 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. Taking advantage of this characteristic the underlying UAS autopilot can be replaced by a flight simulator.

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 waypoints. These waypoints feed the VAS. When condition results or flight plan updates are received, the FPM recomputes all affected waypoints. If invalidated waypoints have already been sent to the VAS, a message is sent informing that they must be discarded.

The Ground Control station consists in a number of consoles that enable interaction with the embarked services. This is the interface the UAS pilot interacts with. In practice, the human pilot is being removed from the aircraft and placed on-ground. He/she will still interact with the ATC authorities and remotely operate the aircraft but, since UAS maneuvers are highly automated, the piloting will mainly consist in sending simple commands to the FPM.

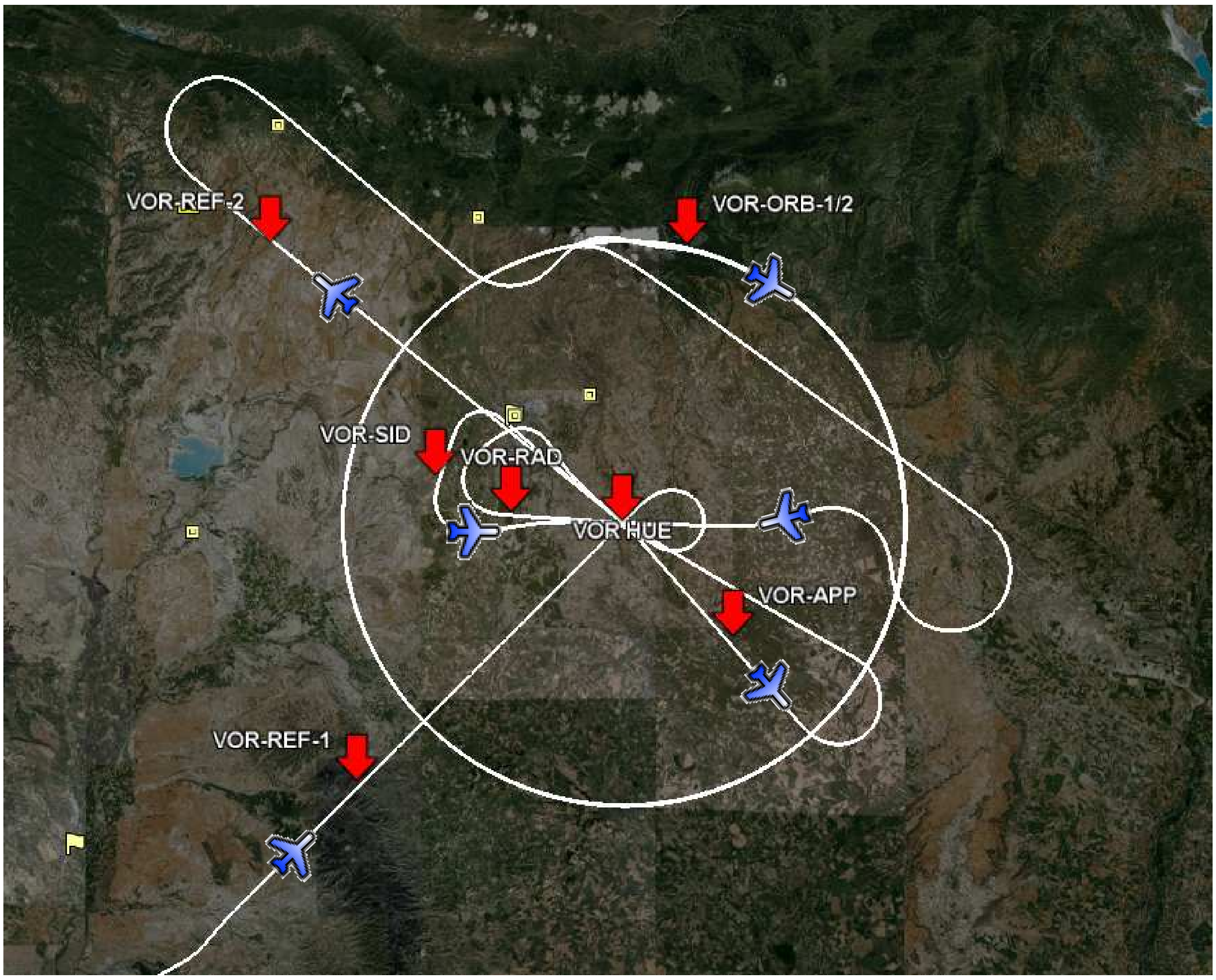


Figure 6.10: *Flight path of complete inspection.*

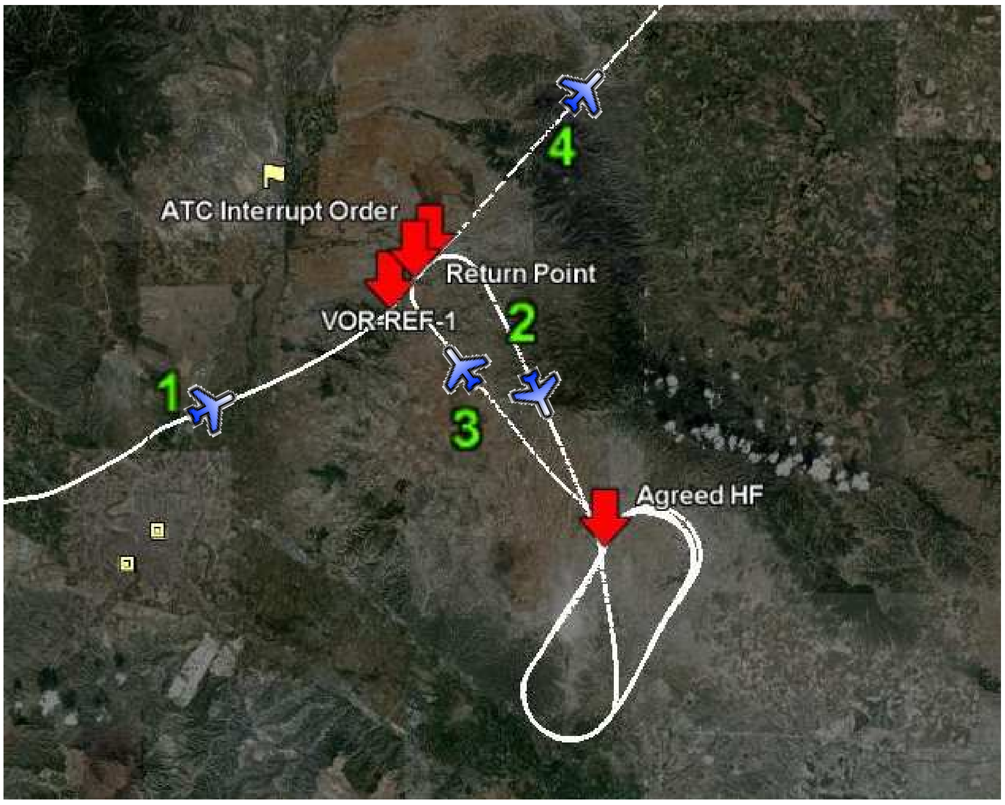
The Flight Tracking System consists in a service that continuously listens to flight data from the VAS and the FPM. This data is passed on to a web server. To display the mission evolution Google Earth® is used. This virtual globe application continually queries the UAS position to the web server and shows its position and trajectory in real time. Images obtained from Google Earth® provide the basis for the figures used to present the simulation results.

6.4 Experimental Results

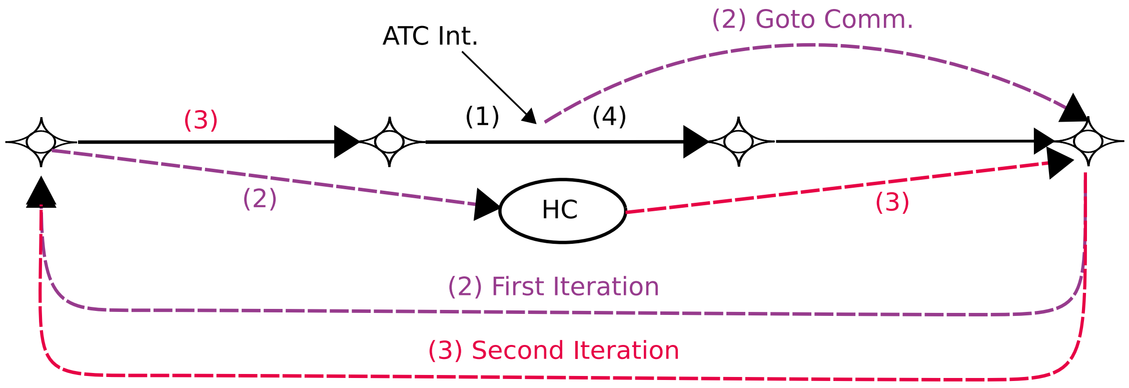
Section 6.2 discusses the different procedures required to perform our example flight inspection and shows how these procedures are coded using the XML based specification language presented in Chapter 4. This section shows the results obtained in the execution of the flight inspection using our simulation environment.

The first test consists in a complete execution of the inspection procedures. The resulting flight is displayed in Figure 6.10. Labeled arrows indicate which procedure different parts of the flight belong to. This is a long flight plan and the simulation confirms that it was well defined and executed.

Next examples relate to the achievement of the agility requirements of the inspection mission. In other words, they show how the UAS is able to interrupt and later on correctly resume the flight



(a) *Aircraft trajectory.*



(b) *Execution steps.*

Figure 6.11: *ATC interruption.*

inspection.

Figure 6.4 shows how the system responds to a situation where an ATC makes a request like: "Fly to X point and hold until further notice". The flight trajectory in Figure 6.11a is numbered to indicate the chronological order of the different steps of the maneuver. Small downward arrows mark some relevant points. In "1" the UAS is initiating the flight of the *VOR-REF-1* procedure (Reference Radial Flight). At the *ATC Interrupt Order* arrow an interruption is requested and the vehicle leaves its trajectory to fly the agreed holding racetrack (step "2"). The position of this holding procedure (*Agreed HF* arrow) can be planned with ATC before the mission execution or decided in real time sending a flight plan update to the FPM. Once the ATC decides that the inspection mission can continue, the UAS returns to the beginning of the leg that was interrupted (step "3") to perform its execution and then proceed with the rest of the flight (step "4").

Figure 6.11b shows how the behavior displayed in Figure 6.4 is supported at the flight plan level. Solid arrows represent the actual legs being flown, dashed arrows indicate jumps or transitions with no associated physical trajectory. The numbering appearing in the figure corresponds to the phases of Figure 6.4. ″1″ represents the leg being flown when the interrupt request is received. Two steps need to be performed by the on-ground operator to command the UAS to execute the holding pattern (step ″2″).

1. Set a condition result so that the chosen path at the next iteration corresponds to the holding pattern (HC).
2. Issue a *Goto* command to jump directly to the end of the iterative leg.

When ATC decides that the inspection can continue, the same steps are repeated but with a different purpose (step ″3″). Now condition results are set for leaving the holding pattern and ensure the branch containing the inspection legs is taken.

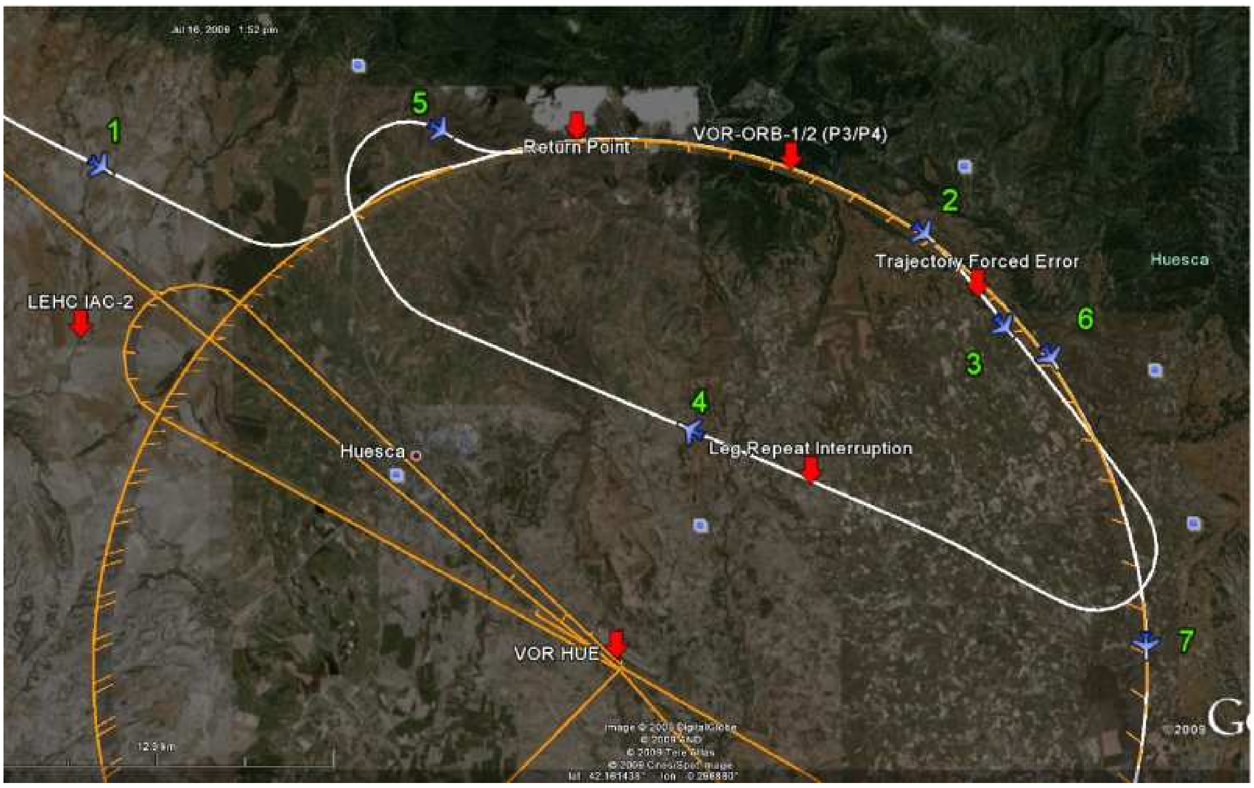
Figure 6.4 displays a situation where a leg needs to be repeated due to a deviation of the planned trajectory. This trajectory error has been manually induced for the purpose of the simulation. As the figure shows, initially the UAS is flying to intercept *VOR-ORB-1* procedure (step ″1″). It flies properly (step ″2″) until a trajectory error occurs (step ″3″), hence measures are not going to be correct. The leg has to be interrupted and flown again. In step ″4″ we see how the UAS interrupts normal mission execution an flies back to a previous return point (step ″5″). From there on the flight continues as planned (steps ″6″ and ″7″).

From the point of view of the flight plan, the structure for supporting such behavior is shown in Figure 6.12b. The main difference with the previous case is that this time no holding procedure is required. When the operator detects a deviation from the expected trajectory (step ″3″) it issues a *Goto* command to jump directly to the end of an iterative structure (step ″4″). Execution of the iterative leg starts over and the UAS intercepts the procedure at the leg preceding the interrupted one avoiding discontinuities in the acquired data (step ″5″).

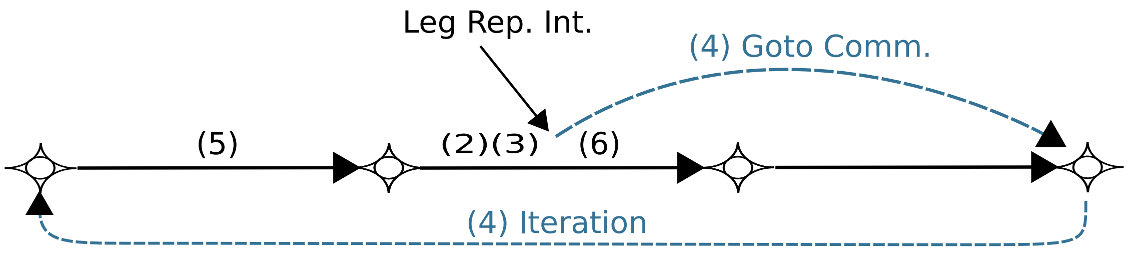
6.5 Conclusion

In this chapter, we have seen how the proposed flight plan specification language, combined with its execution engine, is able to cope with demanding missions like the presented example of a flight inspection. In a mission like this, the agility of the UAS flight operations is a key factor. The simulations show how the system can make use of dynamically set conditions and other commands to jump to any predefined part of the flight plan. With this approach, the UAS can execute retiring maneuvers upon request of ATC. Besides, parts of the inspection procedures can be repeated if deemed necessary by the inspection operator.

The flight inspection mission has been carried out having into account only flight issues. This implies that coordination of flight and payload operation must be done by an onground operator. In the following chapters we will see how the system is extended with an additional layer whose responsibility is to coordinate flight and mission payload. In the inspection example, we could take advantage of this improved level of autonomy by automatically triggering repetition of part of the flight upon detection of an error in sensor readings.



(a) *Aircraft trajectory.*



(b) *Execution steps.*

Figure 6.12: *Leg repetition.*

7 The Mission Manager Service

In Chapter 2, some fundamental notions about autonomous mobile robot architectures have been introduced together with a number of works that tackle mission specification and management for autonomous vehicles.

Our system implements a distributed architecture where basic reactive behaviors are provided by flight and payload services. On top of that, more complex behaviors can be defined by virtue of the Mission Manager (MMa) and State Chart XML (SCXML) (W3C, 2009). The MMa is the embarked service responsible for coordinating operation of UAS services during the mission. SCXML is the language we propose for describing the UAS behavior during the mission.

In this chapter, we firstly introduce the MMa service and the role it plays in our UAS architecture. Afterwards, brief overviews of statecharts, which provide the basis for SCXML, and of SCXML itself are provided. Finally, we see how the selection of SCXML allows us to take advantage of one of its existing implementations for building a Mission Manager prototype.

7.1 The Mission Manager Service

The goal of the MMa is to extend the UAS' autonomous capabilities by being able to execute an specification of the UAS behavior. The specification determines how operation of embarked services is going to be orchestrated in order to perform a given mission. The language chosen for describing the UAS behavior is SCXML.

As shown in Figure 7.1, the MMa listens to events coming from the FPM, the VAS and other services. When such events occur, it reactively responds operating on the FPM and mission related services to control the execution of the mission. While Figure 7.1 emphasizes the

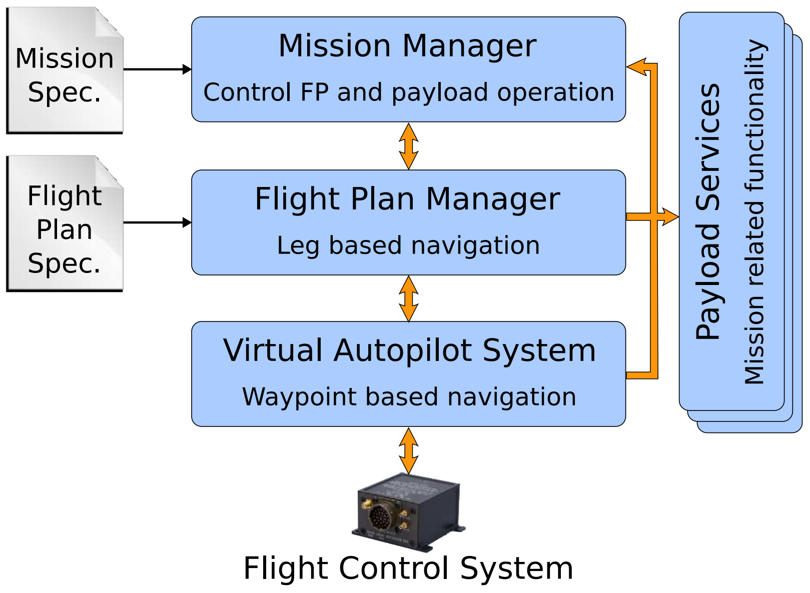


Figure 7.1: *Relationship between MMa, FPM, VAS and other payload.*

hierarchical nature of the relationship between the MMa, the FPM and the VAS, all inter-service communications take place through a shared bus.

From a functional point of view, the MMa supports operations to receive the mission specification, initiate its execution and perform its finalization when operation of the MMa is no longer needed. Apart from that, the number and types of communication primitives handled by the MMa really depend on the mission at hand and the kinds of embarked payload.

While some payload services may vary from mission to mission, the presence of the FPM and the VAS can be taken for granted. Therefore, the MMa subscribes to position, VAS state and other events produced by the VAS, and to leg and stage events produced by the FPM. These messages inform the MMa about the flight progress and allows it to control payload operation in accordance.

Events coming from payload services can also affect the UAS flight, therefore the MMa has the ability to act on the FPM setting condition values, generating *Goto* commands and issuing updates that change the flight plan.

The MMa can interact with the payload services in different ways. As an example, it can start/stop their operation. By doing so, a sensor such as a camera could be turned on only during the period of time when it is actually needed. Other interactions may involve switching modes of operation, e.g., change resolution or interval between snapshots, etc. Specialized mission services need to inform the MMa when relevant events occur, some examples consist in notifying that an element of interest has been detected or that a downlink communication channel has become available.

7.2 Statecharts

Statecharts (Harel & Politi, 1998) can be used to model the behavior of a complex reactive system by means of a finite number of states, transitions between those states, and actions.

- A *state* reflects the current configuration of the system. A pseudo-state graphically represented as a filled dot is used to indicate the state the system is at when its execution starts.

- 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.
- The events that cause a reaction are called *triggers*.
- A *condition* is a boolean expression used to specify under what circumstances a given transition is permitted.
- Transitions can be accompanied by actions to be performed during the transition. An *action* specifies an executable computation. Typical things actions are used for include firing another event, updating some data structure and interact with the outside world. Actions can also be executed when a state is entered or when it is exited.

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.

Statecharts extend traditional state machine diagrams with support for hierarchy and orthogonality, which respectively enable modular descriptions of complex systems and provide constructs for expressing concurrency. Statecharts are part of the Unified Modeling Language (UML) (OMG, 2010; Booch *et al.* , 2005), a widespread graphical modeling language used in industry and academia.

7.2.1 Hierarchical decomposition

Highly complex behavior is difficult to describe with flat diagrams. As the number of states increases the description of the system becomes less and less manageable. Statecharts address this problem by providing a decomposition mechanism that enables the organization of the system description in smaller and easier to manage modules.

Figure 7.2 illustrates the application of hierarchical decomposition. Figure 7.2a shows a composite state *s4* with two inner states *s1* and *s3*. *s1* can be reached from *s2* when event *a* occurs. *s3* can be reached from *s1* when *d* occurs or from *s2* when *c* occurs. If the system is in state *s4* and *b* occurs, it will transition to *s2* regardless of which of the inner *s4* states it is in. In Figure 7.2b the details of *s4* are abstracted away providing a simpler view to work with. Analogously, Figure 7.2c shows the resulting graph when we want to concentrate on *s4*.

It is not mandatory that all inbound arrows of a composite state reach one of its substates. The composite state can contain an initial pseudo-state. In addition, it can contain a history pseudo-state to indicate that the system is able to remember what state it was in the last time the composite state was exited.

Hierarchical decomposition is sometimes referred to as XOR decomposition. In our example, when *s4* is reached, the system must be in *s1* or *s3*, but not in both.

7.2.2 Concurrency

Figure 7.3a illustrates another advanced feature of statecharts, namely orthogonal decomposition. Note how a dashed line divides state *Y* into two separate regions. This notation indicates that when the system is in state *Y*, it must be in some combination of *B* or *C* with *E*, *F* or *G*. This type of decomposition is also referred to as AND decomposition, because being in a state implies being in all of its AND components.

Figure 7.3a contains a number of situations worth mentioning. First, note that event *e3* appears in both states *A* and *D*. When this event occurs, transitions from *B* to *C* and from *F* to

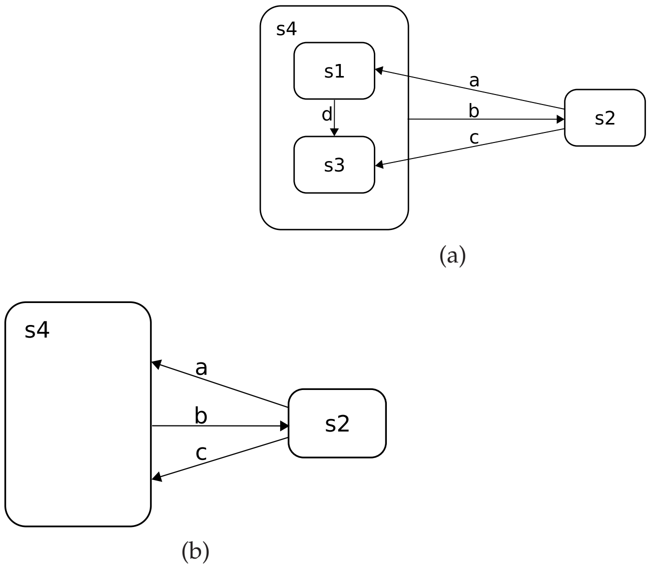


Figure 7.2: *Hierarchical decomposition of states.*

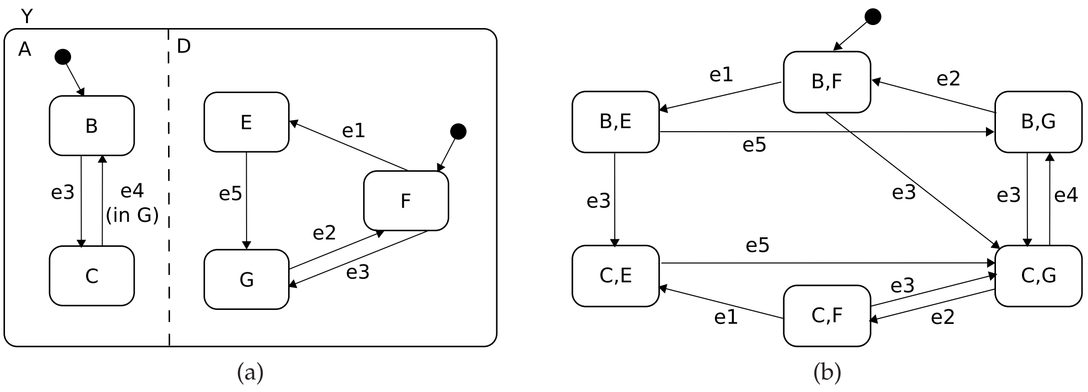


Figure 7.3: *Orthogonal decomposition of states.*

G will simultaneously take place. Other events, like *e1* only affect one of the AND components. Finally, note the condition attached to the transition from *C* to *B*. This transition will take place only if the system is in state *G*. Therefore, while orthogonal decomposition describes independent aspects of the system, common events and guard conditions can be used to provide certain kinds of synchronization.

The concurrency capabilities of statecharts can dramatically reduce the complexity produced by exponential blowup found in traditional state diagrams, where all state combinations need to be considered. This fact is illustrated in Figure 7.3b. The six states found in Figure 7.3b result from the combination of the two states of *A* with the three states of *B*. Any increment in the number of states of *A* or *B* could rapidly lead to a completely unmanageable description if traditional graphs where to be used.

7.3 StateChart XML (SCXML)

SCXML is a working draft published by the World Wide Web Consortium with its latest version released on October 2009. Although it originated as a control language for voice and multimodal interfaces it can also be used in places where reactive control is needed. SCXML provides and XML

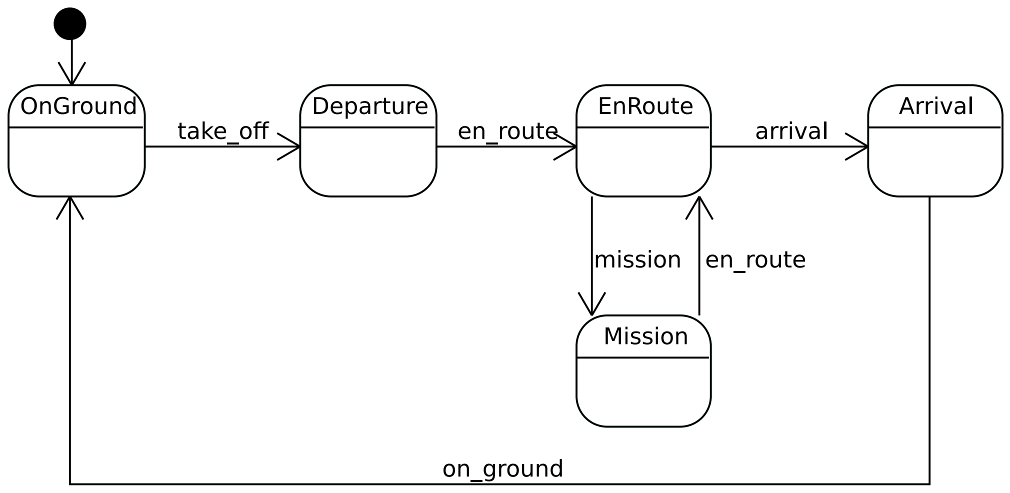


Figure 7.4: *Statechart with main states of a mission.*

syntax that encapsulates the semantics of Harel's Statecharts combined with an XML syntax. In this way statecharts, which have been defined as a graphical specification, can also be represented as text.

As an example, consider the statechart depicted in Figure 7.4. The diagram shows the main states an UAS goes through to perform a certain mission. When the system starts it will enter the *OnGround* state. Changes of state occur as a result of the UAS making progress in the execution of the flight plan. When notification that the UAS is taking off is received the transition to *Departure* takes place. The system remains in this state until the *en_route* event is received. From the *EnRoute* state two different destinations can be reached: *Mission* and *Arrival*. The UAS stays in the *Mission* state while the main mission operations take place. Afterwards, it goes back to the *EnRoute* state. Finally, the *Arrival* state encompasses all arrival and landing procedures specified in the flight plan. Once on ground the system goes back to the initial state. This is a very simple state diagram with none of the statecharts' advanced features. More complex examples can be found in Chapter 8 where this statechart is revisited and refined in order to fully specify a hotspot detection mission.

Listing 7.1 provides the corresponding SCXML representation of the state diagram displayed in Figure 7.4. The SCXML document starts with the XML declaration. It defines the XML version (*1.0*) and the character encoding being used (*UTF-8*). The next line contains the root element of the document, in our example stating that this is an SCXML document. The three attributes found in the root element respectively specify the default namespace of child elements, the SCXML version and the initial state. The *state* tag is used to declare states, each one of them is given a unique id. Inside each stage, transitions with origin at that state can be found. Note that most transitions are triggered by an event called *current_stage*. This event is issued by the Flight Plan Manager to indicate that execution of a new stage has started. To determine what is the current stage, *_eventdata* is checked. The *target* attribute specifies what state a given transition leads to. Note that this differs from the simplified view of Figure 7.4, because most of the events present there really translate to a *current_stage* event plus a condition. It is also worth noting the *src* attribute in the Mission state, whose value is the name of a file containing a submachine that refines this state.

Listing 7.1: SCXML encoding of UAS mission state diagram.

```
<?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="Departure">
      <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=='retroute'" target="EnRoute"/>
      <transition event="terminate" target="EnRoute" />
   </state>

   <state id="Arrival">
      <transition event="on_ground" target="OnGround" />
   </state>

</scxml>
```

SCXML is composed of different modules, each one providing a set of tags and its semantics, that define logical units of functionality. This organization provides more flexibility since it allows applications to select the features they want to support. These are the modules SCXML is organized into:

- Core Module: contains the elements that define the basic Harel state machine. It provides elements to specify states, transitions and some executable content. Executable content consists of actions that are performed as part of taking transitions and entering and leaving states. This module includes, among others, XML tags such as *<scxml>*, *<state>*, *<transition>*, *<parallel>*, *<initial>*, *<final>*, *<onentry>*, *<onexit>*, etc.

- External Communications Module: adds the capability of sending and receiving events from external entities, as well as invoking external services. Tags included in this module are *<send>*, *<cancel>*, *<invoke>* and *<finalize>*.

- Data Module: implements the capability of storing, reading and modifying a set of data that is internal to the state machine. This module provides the tags *<datamodel>*, *<data>*, *<assign>*, *<validate>* and *<param>* along with a number of system variables.

- Script Module: adds scripting capabilities to the state machine. A single tag called *<script>* is provided.

- Anchor Module: is intended to provide 'go back' or 'redo' functionality that is useful in some applications. *<anchor>* is the only tag provided by this module.

Specific details regarding each one of these modules can be found in (W3C, 2009). An interesting feature of SCXML is that it can be extended with custom actions, meaning that new user defined tags can be added as executable content.

7.4 Algorithm for SCXML interpretation

This section provides a simplified view of the normative algorithm that accompanies the SCXML specification draft. As shown in Figure 7.5 the SCXML execution engine operates on four main data structures:

- The *External Events Queue* receives and stores all external events that reach the statechart.

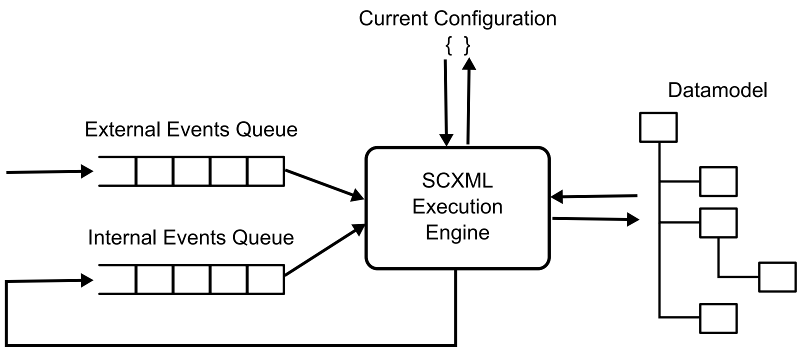


Figure 7.5: *Schematic view of the SCXML execution engine.*

- The *Internal Events Queue* receives and stores internal events.
- *Current Configuration* maintains the state or list of states the statechart is in.
- *Datamodel* is used to store data.

The execution engine follows a step-by-step execution model. All external events go through the external events queue. The execution engine's main loop takes one external event at a time and performs its execution in a run to completion fashion, which means that all pending events will wait until execution of the current one has finalized. This is referred to as a macrostep.

The actual execution of the event is performed in so-called microsteps. During a microstep all transitions enabled by an external event are processed. This involves:

1. Execute content for all abandoned states that have actions to be run on exit.
2. For each enabled transition run its executable content.
3. Run executable content that needs to be executed when entering target states and update the statecharts's configuration.

During the microstep execution, the datamodel may have been modified as a result of running executable content. Also internal events may have been raised. These events are stored in the internal events queue and are processed in a subsequent microstep execution. A macrostep is completed when the internal queue becomes empty. At that point another event from the external queue will be processed.

As an example, consider the state machine of Figure 7.6, suppose that events *e1* and *e2* are in the external queue and that the current configuration is $\{A\}$, meaning that the state machine is in state *A*. When processing *e1*, during the transition to *B*, an internal event *e3* will be raised, which gets stored in the internal queue. Since all events in the internal queue must be processed before executing the next macrostep, the system is guaranteed to transition to state *D*. This happens even though *e2* was already in the external event queue when *e3* was generated.

The execution algorithm makes use of priorities based on document order to prevent ambiguous situations. An SCXML statechart that does not invoke any external event processor must always react with the same behavior to a given sequence of input events.

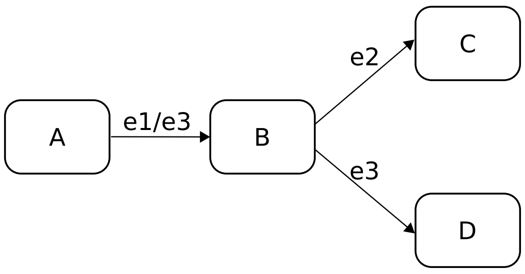


Figure 7.6: *State machine example to illustrate processing order of events.*

7.5 Implementation of MMa Prototype

A big advantage 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. There are at least three open source projects that implement a generic event-driven SCXML based execution environment: a C++ version that integrates with Nokia's Qt framework (Nokia, 2010), a Java version being developed as part of the Apache Commons project (The Apache Software Foundation, 2010) and a Python version from independent developers (Lager, 2010) which seems to be the less mature of the three.

Figure 7.7 shows a diagram of the main classes participating in the implementation of the MMa. In our prototype Commons SCXML is used as the execution engine that lies at the core of the service. Some additional classes wrap around the SCXML engine and handle communications with other services.

We now proceed to describe the responsibilities of each class starting with those that are provided by Commons SCXML :

- *SCXMLExecutor*: implements the execution algorithm described in section 7.4.
- *SCXML*: holds the internal representation of the SCXML mission specification once it has been parsed.
- *JexlContext*: is the place where SCXML datamodels are stored. The MMa service can access the data that the SCXML engine operates with through context objects. *JexlContext* is a particular implementation of *Context* fitted to the Jexl language.
- *JexlEvaluator*: all expressions present in the SCXML document need to adhere to a certain language syntax. Jexl is one such language and *JexlEvaluator* is the class used to evaluate expressions written in Jexl.
- *EventDispatcher*: provides the interface definitions that any of its implementations must honor. If an *EventDispatcher* object is passed to the *SCXMLExecutor*, it will be called every time the execution engine generates an external event. Therefore it can be used to bridge the execution engine with the runtime environment.
- *TriggerEvent*: represents an external event that can be passed to the *SCXMLExecutor*.

The rest of classes that appear in Figure 7.7 are not directly provided by Commons SCXML. They are brand new classes or classes that implement an existing interface, whose function is to bridge the SCXML execution engine with the rest of the system. A brief description of each one follows:

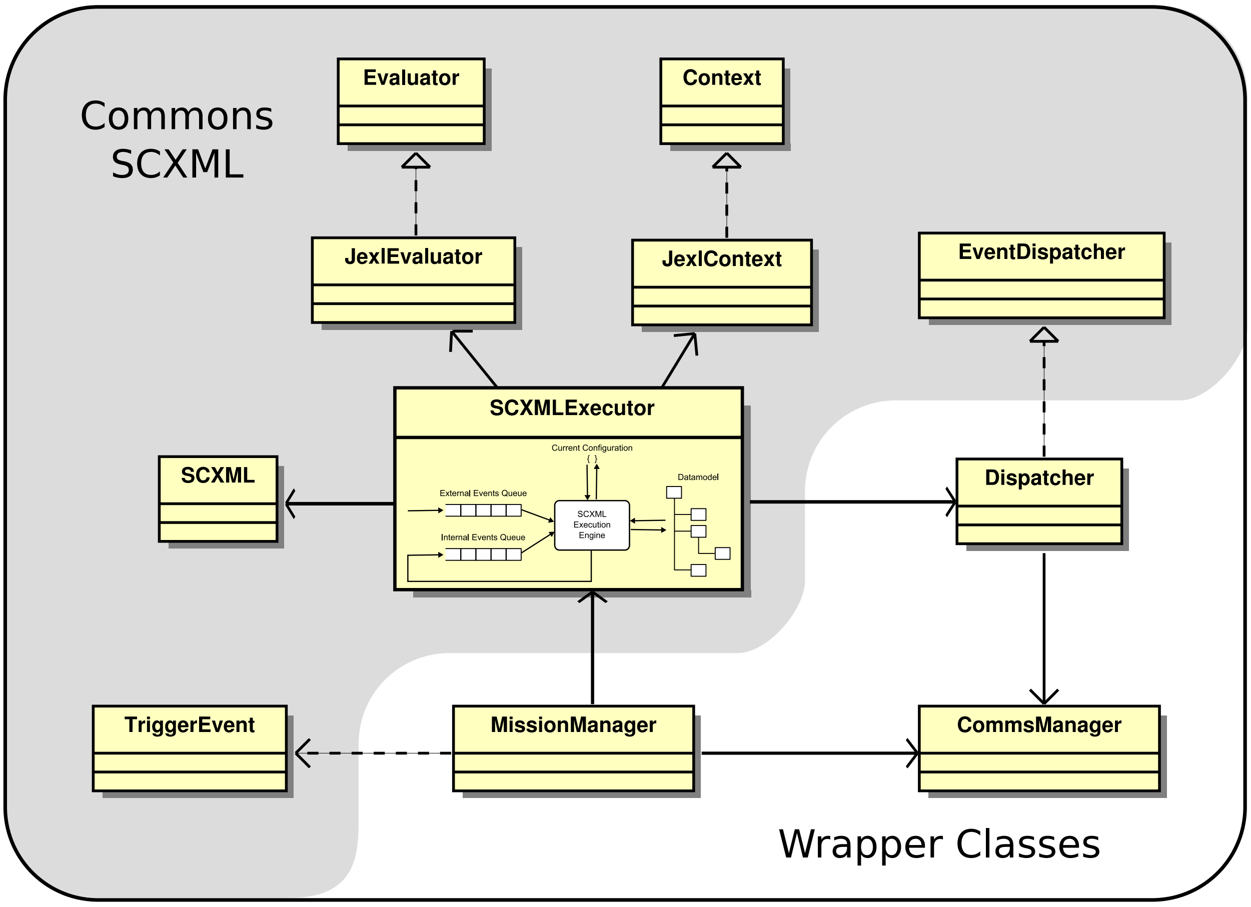


Figure 7.7: *Internal architecture of the MMa prototype.*

- *MissionManager*: is the class that performs all initializations and starts execution of the SCXML engine. It also receives and processes inbound messages. Messages that need to be passed on to the SCXML engine are translated to a suitable representation and forwarded in the form of a *TriggerEvent* object.

- *CommsManager*: is the class that manages network communications. The *MissionManager* subscribes to this class in order to receive notification of incoming messages. The *Dispatcher* uses this class to send outbound events.

- *Dispatcher*: implements an *EventDispatcher*. It gets called by the execution engine when an external event is generated.

The SCXML specification allows implementations to support multiple expression languages to enable using SCXML documents in varying environments. Commons SCXML currently provides support for Commons JEXL and Commons EL (JSP 2.0 EL). While these are similar languages the latter focuses on web development and follows the JSP specification. An advantage of Jexl is its better support for calling methods on Java objects.

7.6 Conclusion

Previous sections have presented our proof-of-concept MMa prototype, but additional work needs to be done in order to turn this prototype into a first class service. As previously stated, the

MMa implements a wrapper around the SCXML execution engine that enables communications with the runtime environment. Since the MMa operates in the context of a flexible system where interacting components may vary, the messages being interchanged with these components can also vary.

In its current form, every new message the MMa needs to receive requires the *MissionManager* class to be updated in order to implement its processing and translation to a friendly form for the SCXML engine. The dispatcher also needs to be updated for every new outbound message. In future iterations, the MMa should be rearchitected to provide a plugin based system where incremental additions could be done without changing previously existing code.

The system communications infrastructure that enables interaction between embarked services follows a publish/subscribe model. It would not make sense for the MMa to try to publish and subscribe to everything. Apart from the aforementioned plug-in capabilities, the MMa should also be able to analyze the SCXML mission, extract the events that are actually used during the mission and publish/subscribe only to them.

Nevertheless, the current implementation is operational and can be used to validate the proposed approach to mission management. This validation is done by means of a simulated hotspot detection mission as explained in the next chapter.

8 Mission Management Experimental Results

In this chapter, we provide the results of a hypothetical hotspot detection mission. The mission consists in flying above the area burned by a wild fire once it has been suppressed. The goal of the mission is to detect remaining hotspots which could revive the fire. Automation of this type of mission can be highly beneficial as it would permit minimizing the resources allocated to this task. As a result, costs would be reduced and valuable resources could be moved to other sites.

8.1 Hotspot Detection Mission

To perform the hotspot detection mission, the burned area will be scanned following a lawnmower pattern. During the scan, imagery is taken that is then processed in order to detect potential hotspots. To determine whether a potential hotspot represents a real threat, each one of them is further analyzed by flying an eight pattern over it.

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.

To exemplify the flexibility provided by separating the specification of the mission from its execution the hotspot detection mission is performed in two ways, both using the same flight plan. In the first case, it is assumed that some time is required to process the recorded imagery. Therefore, there is a delay from the point in time when an image is taken to the moment when it is determined that it contains a potential hotspot. The strategy for performing the mission, in this

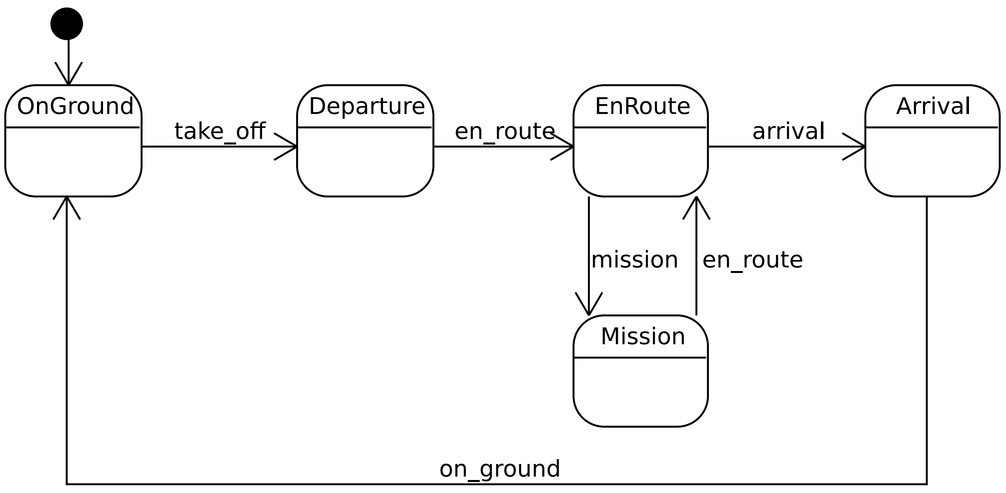


Figure 8.1: *Mission main states.*

case, consists in executing the full scan first and, afterwards, visit each one of the potential hotpots. In the second case, we assume that more capable embarked services are able to detect potential hotspots immediately. Taking advantage of them, 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.

The overall mission plan, which includes the main states the UAS goes through to complete the mission, is shown in Figure 8.1. Unsurprisingly, an almost direct mapping between flight plan stages and mission states can be established. The diagram provides a simplified scheme that will suffice for the purpose of the demonstration. This statechart is common to both versions of our example mission. The differences between the two versions appear when the *Mission* state gets refined. These refinements are explained in Section 8.3.

8.2 Underlying Flight Plan

The underlying flight plan is common to both versions of the mission. Figure 8.2 illustrates what legs belong to the *Mission* stage of the flight plan and how they are organized. The *Mission* stage is executed during the *Mission* state of the statechart of Figure 8.1. During the *Mission* stage the UAS can either perform an scan of the area (*scanArea* leg), an eight pattern (*scanPoint* leg) or a holding pattern (*hold* leg). Which leg is selected depends on the condition of an intersection leg called *patternSelect*. If the result of the condition is 0 *scanArea* is selected, *scanPoint* is selected if its value is 1 and the holding pattern if it is 2. The three alternatives converge at another intersection leg called *join*. Finally an iterative leg called *loop* is used to enable the UAS to alternate between the different options. The different SCXML descriptions of the *Mission* state will result in the MMa communicating and interacting with the FPM in different ways to achieve the two behaviors previously described.

The complete encoding of the *Mission* stage using our XML specification language is provided in Listing 8.1.

Listing 8.1: XML encoding of the Mission stage of the flight plan.

```
<stage id="mission" type="Mission">
   <name>Scan area mission</name>
   <description>A scan over the designated area is performed</description>
   <legs>
      <leg id="loop" xsi:type="fp:IterativeLeg">
```

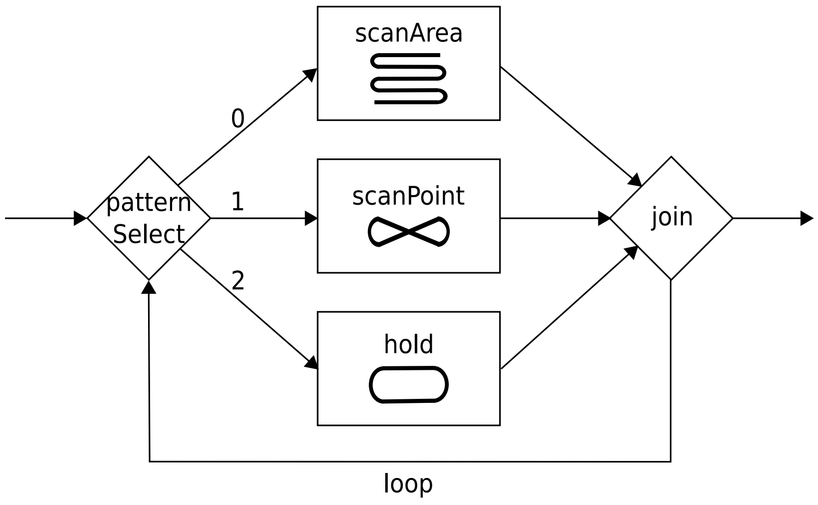


Figure 8.2: *Organization of the legs contained in the Mission stage of the flight plan.*

```
    <body>
      patternSelect scanArea scanPoint hold join
    </body>
    <first>patternSelect</first>
    <last>join</last>
    <upperBound>15</upperBound>
</leg>
<leg id="patternSelect"
    xsi:type="fp:IntersectionLeg">
    <next>scanArea</next> <!-- default value -->
    <nextCond>selection</nextCond>
    <nextList>scanArea scanPoint hold</nextList>
</leg>
<leg id="scanArea" xsi:type="fp:BasicScanLeg">
    <dest>
      <coordinates>
        41.5493424917977 1.77254310685181
      </coordinates>
      <speed>60</speed>
    </dest>
    <next>join</next>
    <dim1>6000</dim1>
    <dim2>5500</dim2>
    <angle>80</angle>
</leg>
<leg id="scanPoint" xsi:type="fp:ScanPointLeg">
    <dest>
      <coordinates>
        41.56947331267459 1.717810982215079
      </coordinates>
      <fly-over>true</fly-over>
      <speed>40</speed>
    </dest>
    <next>join</next>
    <course>135</course>
    <d1>1000</d1>
    <d2>450</d2>
</leg>
<leg id="hold" xsi:type="fp:HFLeg">
    <dest>
      <coordinates>
        41.55523585866938 1.777892046315137
      </coordinates>
      <speed>40</speed>
    </dest>
    <next>join</next>
    <course>45</course>
    <direction>Right</direction>
```

```
            <d1>1000</d1>
            <d2>450</d2>
        </leg>
        <leg id="join" xsi:type="fp:IntersectionLeg" />
    </legs>
    <initialLegs>loop</initialLegs>
    <finalLegs>loop</finalLegs>
</stage>
```

8.3 Refinements of the Mission State

This section describes how, taking advantage of the hierarchical decomposition supported by statecharts, the *Mission* state can be refined in different ways to implement a deferred and an immediate analysis strategy.

8.3.1 Deferred Hotspot Analysis

In this section we discuss the so-called Deferred Hotspot Analysis version of our example mission. In this version we assume that some time is required for analyzing the collected samples and decide that a given location should be analyzed in more detail. The expected behavior is to fully scan a rectangular area first, and visit each one of the potential hotspots afterwards.

Figure 8.3 shows the statechart that refines the *Mission* state. 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 systematically sweeps 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 *HotSpotsCounter* 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 8.3 there are three states which directly map to the corresponding flight plan legs. The *ScanArea* is the 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 theses 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.

For example, when entering *ScanArea* we set the selection condition to 2, meaning that *hold* is going to be our default leg. This behavior is shown in Figure 8.4 and is done just after activating

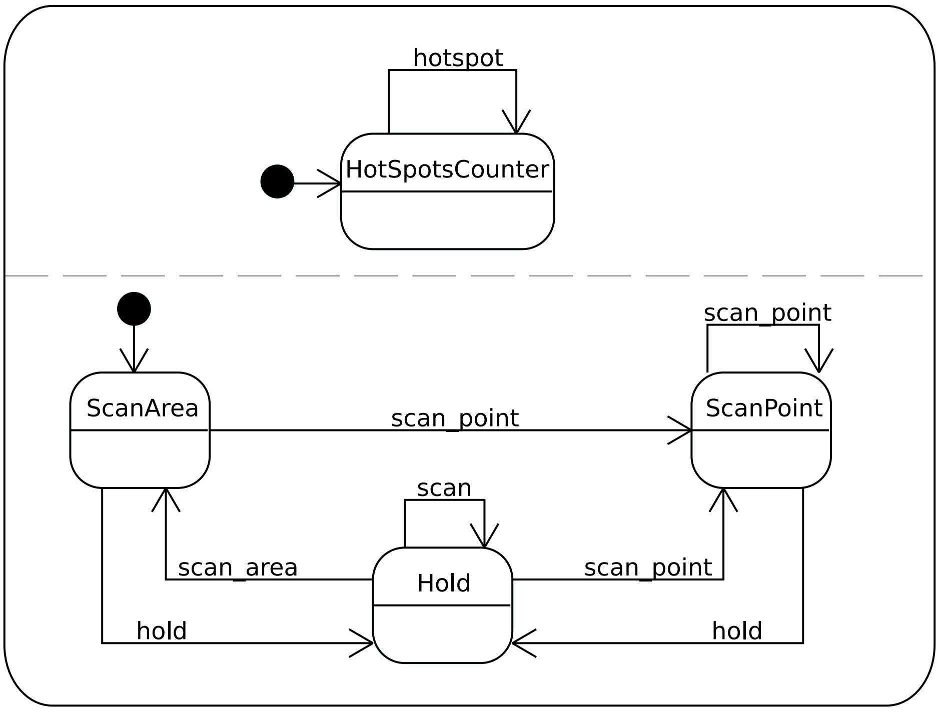


Figure 8.3: *First version the Mission state: hotspot analysis is deferred.*

the payload services required during the *Mission* state. If there is a hotspot that needs to be explored the *HotSpotsCounter* self-transition is triggered and the following actions, also shown in Figure 8.4, 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. Note that, again, the first thing we do when entering the *ScanPoint* state is to prepare the flight plan so that the FPM knows what to do when the current leg finishes. In this state, the selection of the default leg depends on the number of remaining unvisited hotspots. The system will remain in the *ScanPoint* state until all potential hotspots have been visited. Once all hotspots have been visited, 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. If no further scanning is required, the system can abandon the mission area and follow the returning route. At this point, services required only during the *Mission* state can be shut off.

To give an idea of how this behavior is translated to an SCXML document, Listing 8.2 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 8.4 (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

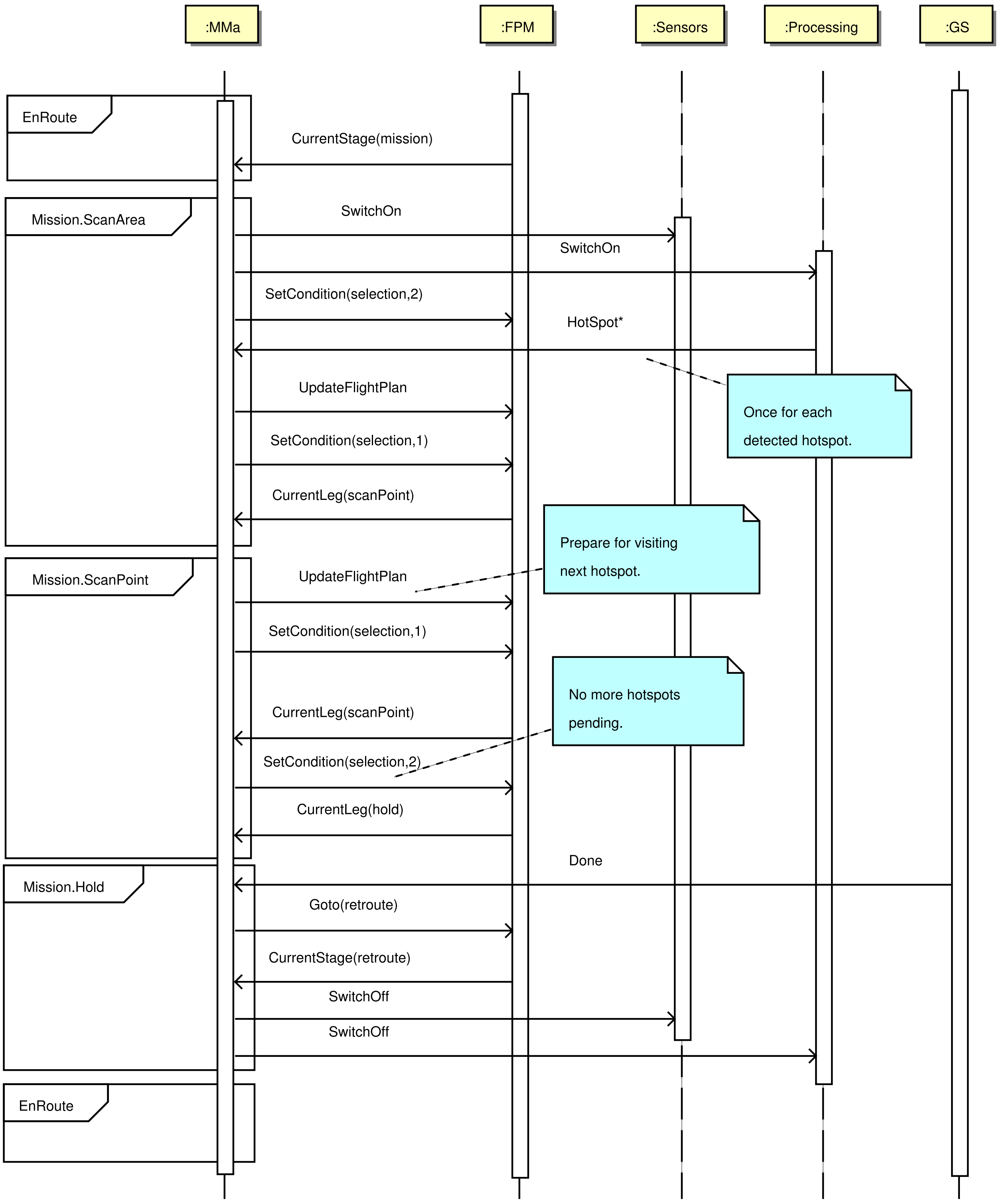


Figure 8.4: *Messages interchanged when performing deferred analysis mission.*

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 8.2: 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()" />
         <assign name="lon"
            expr="HotSpotList[visit_hs_count].getLongitude().toString()" />
         <assign name="coordinates" expr="lat.concat(' ').concat(lon)" />
         <assign
          xmlns:fpu='http://icarus.upc.es/schema/FlightPlanUpdate/1.1'
          location="Data(scanPointUpdate,
          'fpu:FlightPlanUpdate/Change/MainFP/stages/stage/legs/leg/dest/coordinates')"
          expr="coordinates" />
         <!-- Send message to fpm -->
         <send target="'container'" targettype="'x-marea'" event="'update_cmd'"
            namelist="scanPointUpdate" />
         <!-- Set scanPoint as next leg -->
         <assign name="selection" expr="1" />
         <send target="'container'" targettype="'x-marea'" event="'set_condition'"
            namelist="selection" />
      <else/>
         <!-- Set hold as the default next leg -->
         <assign name="selection" expr="2" />
         <send target="'container'" targettype="'x-marea'" event="'set_condition'"
            namelist="selection" />
      </if>
   </onentry>

   <transition
      event="current_leg" cond="_eventdata=='hold'" target="Hold" />
   <transition
      event="current_leg" cond="_eventdata=='scanPoint'" target="ScanPoint" />
</state>
```

Listings 8.3 and 8.4 show the data elements used in the previous SCXML code. Listing 8.3 contains the data elements that are global to the SCXML document and, therefore, can be accessed from anywhere within it. *selection* holds the current value of the condition used in *patternSelect* leg. *detect_hs_count* and *visit_hs_count* respectively store the number of detected potential hotspots and the number of visited ones.

Listing 8.3: Global data elements for managing leg selection.

```
   <datamodel>
      <data id="selection" expr="0" />
      <!-- Detected HotSpots Counter -->
      <data id="detect_hs_count" expr="0" />
      <!-- Visited HotSpots Counter -->
      <data id="visit_hs_count" expr="0" />
   </datamodel>
```

Listing 8.4 contains data elements that are local to the *ScanPoint* state (not included in Listing 8.2 for brevity). Some of them are just temporary variables that do not maintain any kind of state information. The *scanPointUpdate* data element is of special interest because it provides an skeleton of the XML code sent for updating the flight plan. 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 *upd_cmd* message.

Listing 8.4: Data elements used in scanPoint leg updates.

```
<datamodel>
   <!-- Update message for scan point -->
   <data id="scanPointUpdate"
      xmlns:fpu='http://icarus.upc.es/schema/FlightPlanUpdate/1.1'
      xmlns:fp='http://icarus.upc.es/schema/FlightPlan/1.1'
      xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'>
```

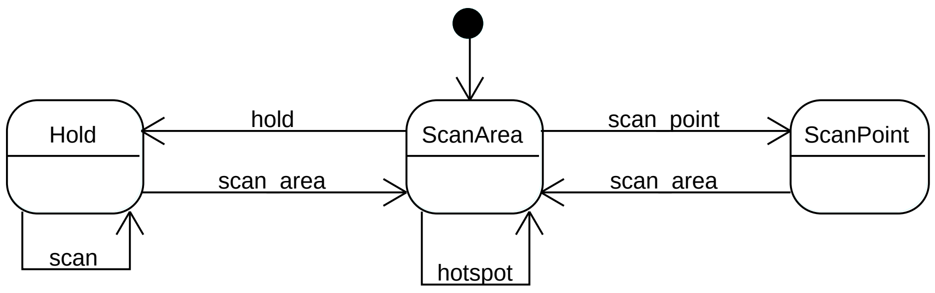


Figure 8.5: *Second version of the Mission state: hotspots are analyzed immediately.*

```
        <fpu:FlightPlanUpdate xmlns="">
        <Change>
           <MainFP targetId="HotSpotPlan">
              <stages>
                 <stage targetId="mission">
                    <legs>
                       <leg targetId="scanPoint" xsi:type="fp:ScanPointLeg">
                          <dest>
                             <coordinates />
                          </dest>
                       </leg>
                    </legs>
                 </stage>
              </stages>
           </MainFP>
        </Change>
        </fpu:FlightPlanUpdate>
     </data>
     <!-- Auxiliary variables -->
     <data id="lat" expr="" />
     <data id="lon" expr="" />
     <data id="coordinates" expr="" />
</datamodel>
```

8.3.2 Immediate Hotspot Analysis

In the Immediate Hotspot Analysis implementation of the mission we assume 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 this case the *Mission* state is refined as shown in Figure 8.5. 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 the previous case. The first thing we do on entering a state is setting a default next leg by changing the selection condition of *patternSelect*. As shown in Figure 8.6, 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 MMa does the following (see Figure 8.6):

  1. Update the *scanPoint* leg with the coordinates of the potential hotspot that must be

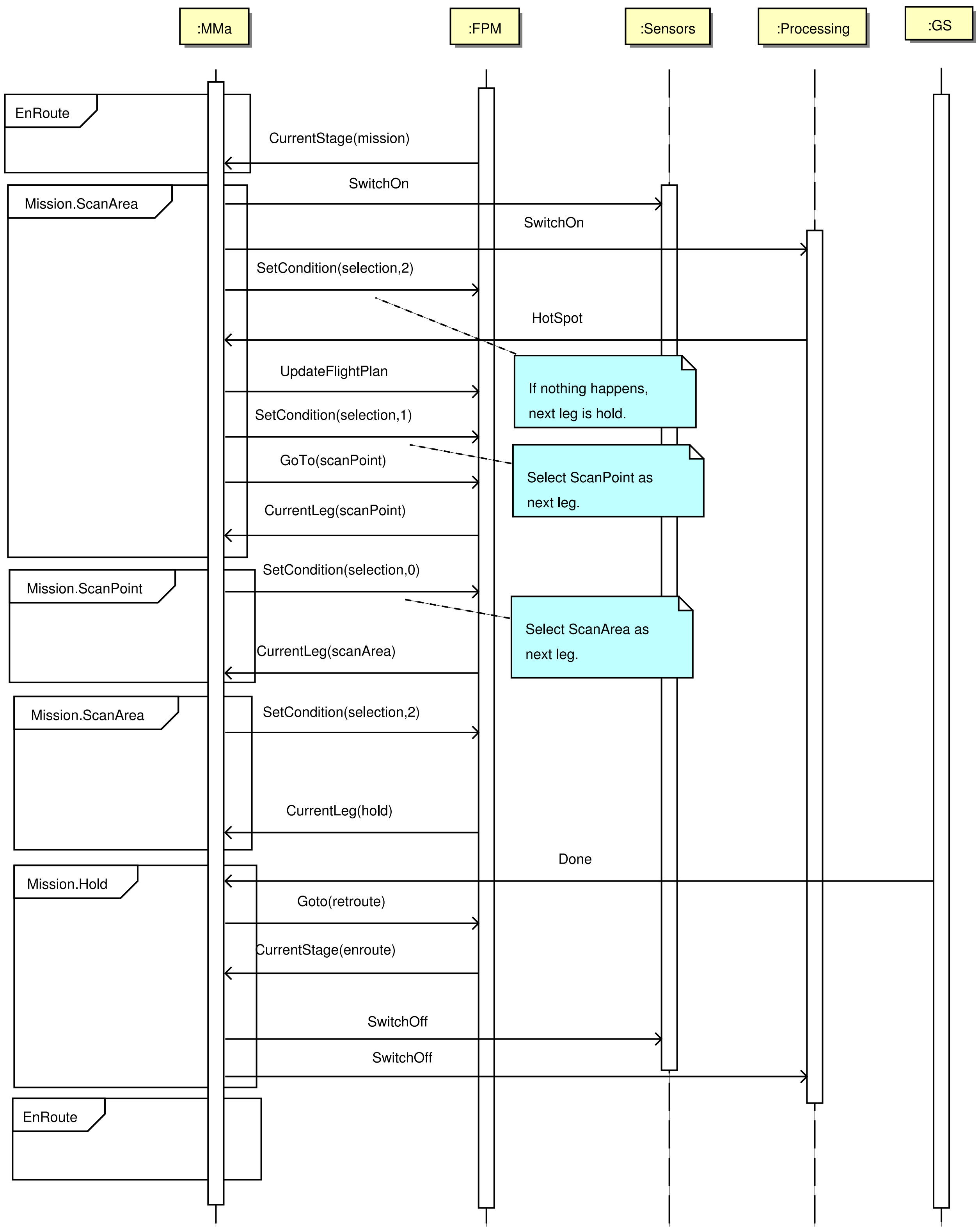


Figure 8.6: *Messages interchanged when performing immediate analysis mission.*

analyzed.

2. Set the result of the selection condition to 1, i.e. select *scanPoint* as the next leg.
3. 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 will trigger the transition to

the *ScanPoint* state.

4. Update the *startAt* parameter of the *scanArea* leg with the position where it has been interrupted so that later on it can be resumed from there.

- *scan_point*: 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.

Listing 8.5 shows the different actions taking place in the *ScanArea* state. Aircraft position data, is provided by the MMa. The location of a potential hotspot comes as a parameter of the corresponding event and is accessed using *_eventdata*.

Listing 8.5: SCXML encoding of ScanArea state.

```
<state id="ScanArea">
   <onentry>
      <!-- When done go to hold state -->
      <assign name="selection" expr="2" />
      <send target="'container'" targettype="'x-marea'"
         event="'set_condition'" namelist="selection" />
   </onentry>

   <transition event="hotspot">
      <!-- Get current position -->
      <assign name="lat" expr="Position.getLatitude().toString()" />
      <assign name="lon" expr="Position.getLongitude().toString()" />
      <assign name="startAtCoords" expr="lat.concat(' ').concat(lon)" />

      <!-- (1) Update scanPoint leg with hotspot position data -->
      <log expr="'HotSpot location: ' + _eventdata['coordinates']"/>
      <log expr="'HotSpot course:   ' + _eventdata['course']"/>
      <assign xmlns:fpu='http://icarus.upc.es/schema/FlightPlanUpdate/1.1'
         location="Data(scanPointUpdate,
         'fpu:FlightPlanUpdate/Change/MainFP/stages/stage/legs/leg/dest/coordinates')"
         expr="_eventdata['coordinates']" />
      <!--  Send message to fpm -->
      <send target="'container'" targettype="'x-marea'"
         event="'update_cmd'" namelist="scanPointUpdate" />

      <!-- (2) Select scanPoint as next leg -->
      <assign name="selection" expr="1" />
      <send target="'container'" targettype="'x-marea'"
         event="'set_condition'" namelist="selection" />

      <!-- (3) Jump to scanPoint leg -->
      <assign name="goto_dest" expr="'scanPoint'" />
      <send target="'container'" targettype="'x-marea'"
         event="'goto_leg'" namelist="goto_dest" />

      <!-- (4) Update startAt parameter of scan so that
         flight is resumed where it was left  -->
      <assign xmlns:fpu='http://icarus.upc.es/schema/FlightPlanUpdate/1.1'
         location="Data(scanUpdate,
         'fpu:FlightPlanUpdate/Change/MainFP/stages/stage/legs/leg/startAt')"
         expr="startAtCoords" />
      <!--  Send message to fpm -->
      <send target="'container'" targettype="'x-marea'"
         event="'update_cmd'" namelist="scanUpdate" />
   </transition>

   <transition event="current_leg" cond="_eventdata=='hold'" target="Hold">
      <!-- Update startAt parameter so that scan starts over from the beginning -->
```

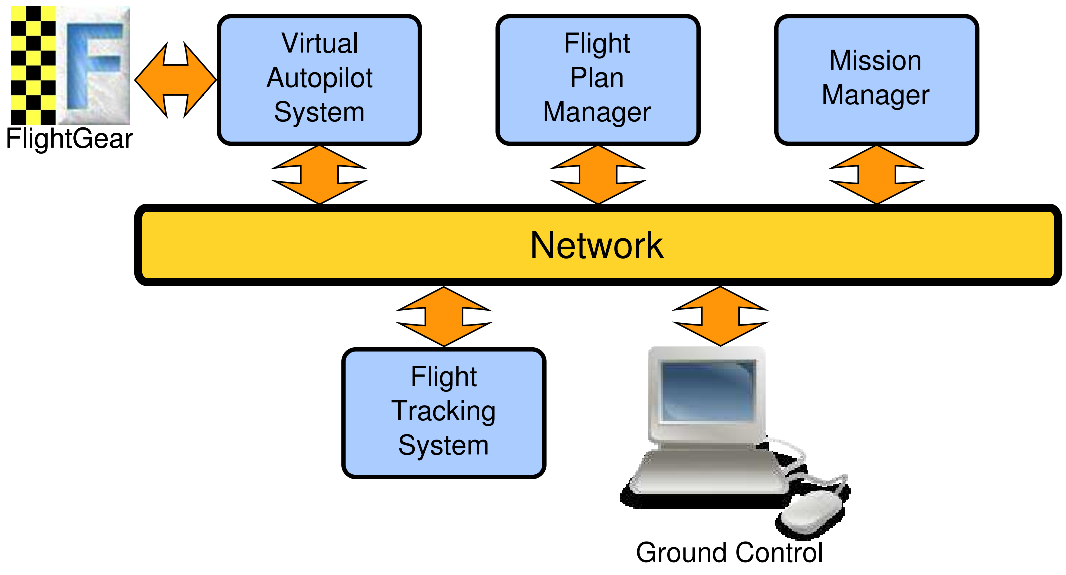


Figure 8.7: *Simulation environment.*

Aircraft	Piper J3 Cub
Cruise speed	65kt (120 km/h)
Bank angle	20 °
Roll factor	3 s
Wind	no wind
Fuel	unlimited
Mission duration	1 h aprox.

Figure 8.8: *Aircraft and simulation parameters.*

```
        <assign xmlns:fpu='http://icarus.upc.es/schema/FlightPlanUpdate/1.1'
            location="Data(scanUpdate,
            'fpu:FlightPlanUpdate/Change/MainFP/stages/stage/legs/leg/startAt')"
            expr="''" />
        <!-- Send message to fpm -->
        <send target="'container'" targettype="'x-marea'"
            event="'update_cmd'" namelist="scanUpdate" />
    </transition>

    <transition event="current_leg" cond="_eventdata=='scanPoint'"
        target="ScanPoint" />
</state>
```

8.4 Simulation Environment

As shown in Figure 8.7, the simulation environment from Section 6.3 is now extended with the addition of the Mission Manager (MMa) service. Again, boxes above the network bar represent embarked components, while boxes below the network bar belong to the ground segment.

FlightGear's aircraft model used in this simulation is a Piper J3 Cub (see Figure 8.8). Experimentally it has been determined that the bank angle used by this model for turning is 20 degrees. A roll factor of 3 seconds accounts for the time it takes to reach the bank angle. The simulation has been run without wind and with unlimited fuel.

The Virtual Autopilot System (VAS), which amongst other features provides waypoint navigation capabilities, and the Flight Plan Manager (FPM), that interacts with the VAS to control the UAS flight, have been extensively discussed in previous chapters. Next to these services the MMa is added. A prototype has been implemented that reads in the SCXML based specification of the mission and makes use of Commons SCXML for its execution.

To perform the example mission, the MMa needs to interact with both payload and flight services. In our simulation the focus is placed on the interactions between the MMa and the FPM. The main event that the UAS needs to execute the mission is the notification that a new potential hotspot has been detected. For the time being this event is generated by the MMa prototype at certain prefixed locations. Apart from that, the MMa listens to the aircraft position generated by the VAS and to events informing about the flight progress from the FPM.

We assume the existence of the necessary payload to take images, analyze them and decide whether further inspection of a given point is required. The same mechanics that enable the MMa to communicate with the VAS and the FPM apply to any other service.

For a better comparison of the deferred and immediate analysis versions of the mission the same locations are used for potential hotspots. 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 deferred analysis, since we need not only to know the amount of detected hotspots but also their positions.

As before, the Ground Control station provides different consoles to interact with the embarked services. In the example shown in this chapter the UAS is able to autonomously complete the mission, but real missions are going to be more complex and we expect one or more human operators to continuously supervise the operations of the UAS and intervene if necessary.

The Flight Tracking System operates as described in Section 6.3 and enables us to follow UAS operations in real time using Google Earth®.

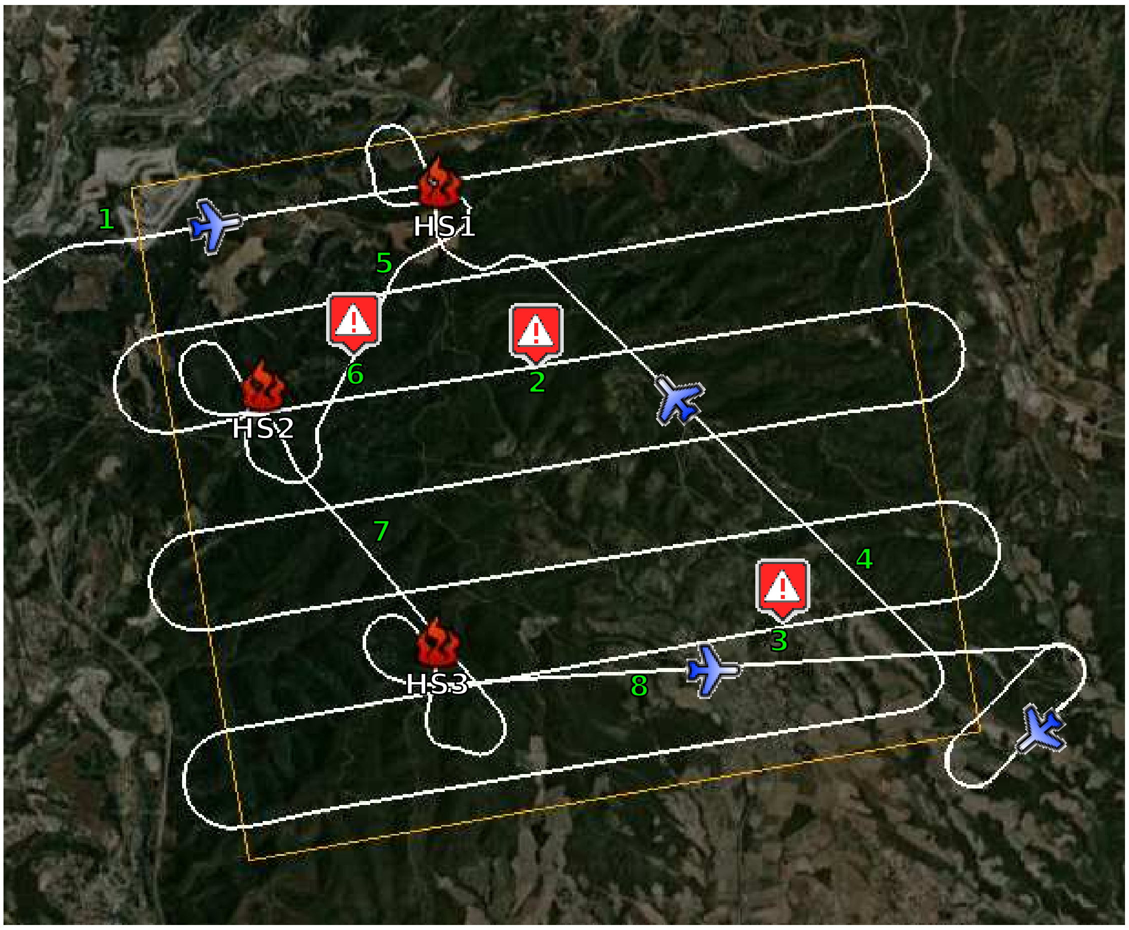
8.5 Experimental Results

The proposed approach for increasing UAS mission automation has been tested using our hypothetical mission and the simulation environment just described. Simulations for both immediate and deferred analysis situations have been executed with three potential hotspots at fixed positions. In both cases, the FPM has been initialized with the flight plan described in Section 8.2.

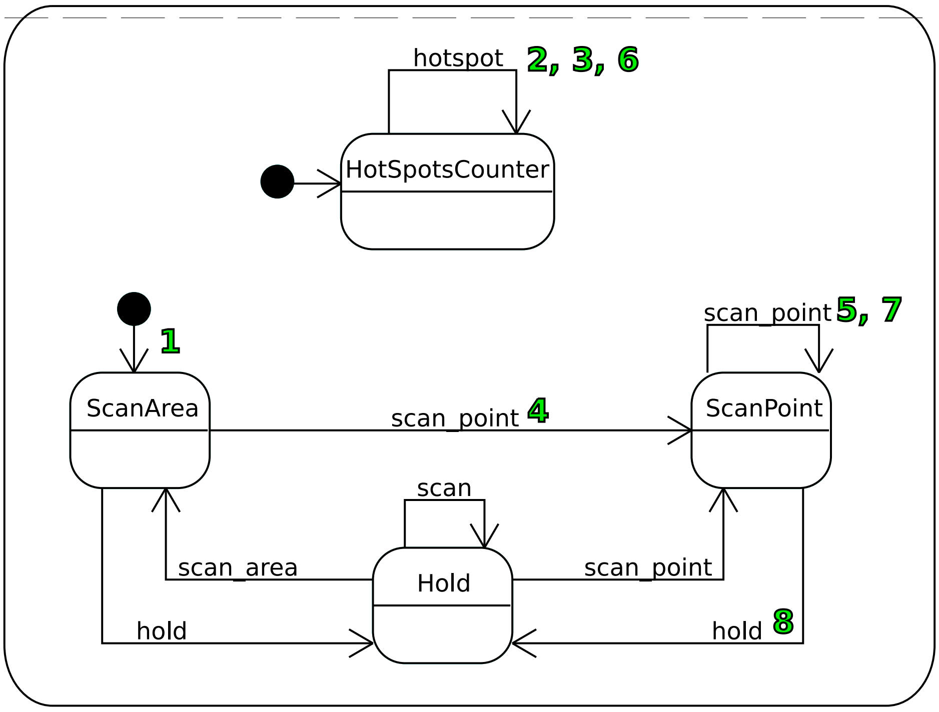
Figure 8.9a shows the trajectory of the aircraft when detection of potential hotspots is not immediate. Bonfire icons along the trajectory indicate the position where the potential hotspots are located. Exclamation mark icons indicate the point in time when each one of the potential hotspots is detected. The different numbers represent relevant events occurring during the execution of the mission. The same numbers are also used in Figure 8.9b, which shows the statechart of the mission, to indicate what transitions are taking place and in what order.

Below, each of the events shown in Figures 8.9a and 8.9b is briefly discussed.

1. At this point, execution of the *scanArea* leg is initiated and, as a direct consequence, the statechart transitions to the corresponding *ScanArea* state. In this simulation, hotspots are



(a) *Aircraft trajectory.*



(b) *State transitions.*

Figure 8.9: *Hotspot mission with deferred analysis.*

only visited once the scan has finished. During this time, the statechart remains in the *ScanArea* state.

2. A little while after overflying the first hotspot, an image processing service realizes that that point should be analyzed. The Mission Manager is notified and this triggers a self-transition on the *HotSpotsCounter* state. During this transition the number of detected potential hotspots is incremented and the corresponding messages are sent to the flight plan to initiate execution of the *scanPoint* leg when the end of the burned area is reached.

3. The second hotspot is detected. Again a self-transition on the *HotSpotsCounter* state is triggered. The number of potential hotspots is increased, but the first one remains as the destination of the *scanPoint* leg.

4. The end of the *scanArea* leg has been reached and execution of the *scanPoint* leg starts. This will take us to the first hotspot. The statechart transitions from the *ScanArea* state to the *ScanPoint* state. During this transition the number of visited hotspots is incremented and the *scanPoint* leg is updated with the coordinates of the second hotspot. These change does not affect the execution of the current instance.

5. An eight pattern has been flown over the first hotspot and now we head for the next one. This triggers a self-transition on the *ScanPoint* state. Since at this time this is the last point to visit, the condition in *patternSelect* is changed to perform the holding pattern once the execution of the current leg finishes.

6. A new potential hotspot is detected. A last self-transition on *HotSpotsCounter* state is triggered. The flight plan is updated so that the new hotspot is visited next.

7. The second and last self-transition on the *ScanPoint* state is triggered. During the transition the number of visited hotspots is increased. Since the number of detected hotspots is equal to the number of visited ones, the flight plan is updated to execute the holding pattern.

8. With the execution of the holding pattern, the statechart transitions to the *Hold* state, where it will remain waiting for further events.

The results of the simulation with immediate potential hotspot detection are shown in Figure 8.10a. As in the previous case, some icons and numbers are overlaid on top of the picture to illustrate what is going on. Figure 8.10b shows the statechart using in this mission. Numbers appearing in both figures indicate events taking place.

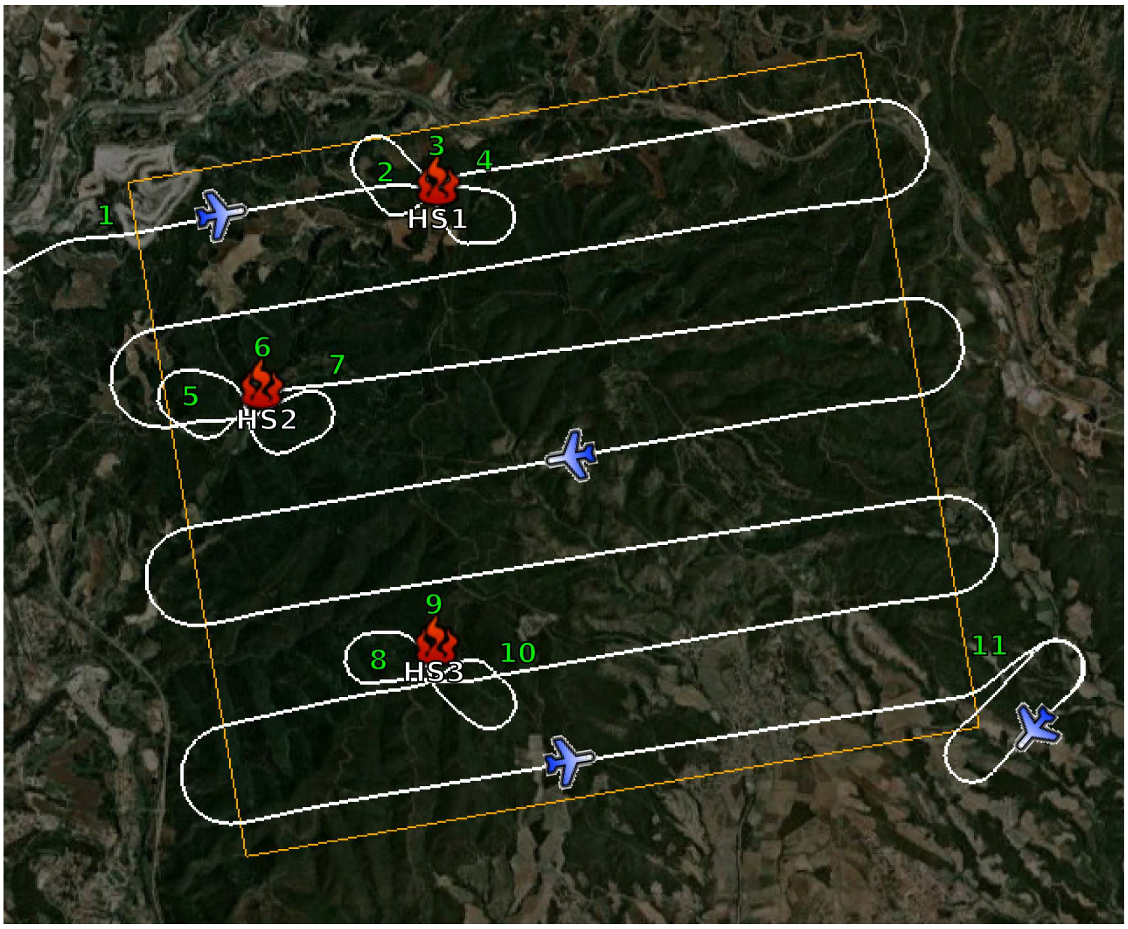
Events shown in Figures 8.10a and 8.10b are treated as follows:

1. At this point execution of the *scanArea* leg starts and the statechart transitions to the *ScanArea* state.

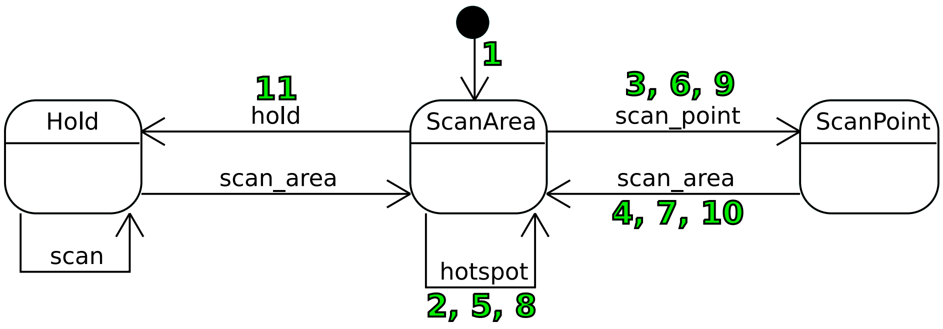
2. A potential hotspot has been detected. This triggers a self transition to the *ScanArea* state where the MMa prepares the execution of the eight pattern. This preparation consists in updating the *scanArea* leg, so that the scan can later be resumed from the current position, updating the *scanPoint* leg with the coordinates of the potential hotspot and, finally, sending a command to the FPM in order to start execution of the *scanPoint* leg.

3. This event is raised when the execution of the *scanPoint* leg starts. The statechart transitions to the *ScanPoint* state. On entering this state, *scanArea* is selected in the flight plan as the next leg.

4. When execution of the eight pattern ends, the FPM seamlessly starts executing the *scanArea* leg, but this time starting from the position where it was interrupted.



(a) *Aircraft trajectory.*



(b) *State transitions.*

Figure 8.10: *Hotspot mission with immediate analysis.*

Events 5,6,7 and 8,9,10 operate in exactly the same fashion as described for events 2,3,4. The only remaining event is number 11, which is a notification that execution of the holding pattern has started. When this event is received, the *scanArea* leg is updated so that if executed again it starts over from the beginning of the area of interest.

8.6 Conclusion

In this chapter, we have seen how by means of using a mission specification separate from its execution engine we are provisioning the UAS with a high level of both flexibility and autonomy.

SCXML has been used as the language for specifying the UAS behavior. Using SCXML the semantics of statecharts are combined with a data model and a scripting language that, put together, provide a very capable method for specifying autonomous behavior.

The performed simulation already reveals the need to complement the reactive execution engine with deliberative capabilities. In particular, the deferred hotspot analysis mission could benefit from a planning service able to generate an optimal traversal of the potential hotspots.

9 Conclusions and Further Work

In recent years, Unmanned Aircraft Systems have been getting much attention. The realization of its potential benefits in the civil domain is fueling many research and development efforts. As a result, a significant number of platforms and autopilot systems are making its way into the market. In the vast majority of cases, the capabilities of the autopilots for these systems are limited to waypoint based navigation, with little support for complex dynamic flight plans. Besides, there is a lack of integration between the aircraft's navigation and payload operation. The main goal of this doctoral thesis is to overcome these limitations and provide a more capable platform, with better support for complex flight plans and an increased level of autonomy. To do so, we propose extending current systems with reconfigurable flight and mission management layers. These new layers provide higher level abstractions for navigation control and enable embarked payload to operate accordingly to the current flight phase and mission needs.

In this chapter, the contents of the thesis are summarized highlighting its main contributions. Afterwards, different aspects that should be studied in more detail as well as some directions for future research are outlined.

9.1 Summary

The main aspects covered in this PhD thesis are:

- A new concept for the specification of UAS flight plans that organizes the flight plan into different phases and makes use of legs as the main unit for its construction. The language design takes into account the current state of the art in UAS systems and commercial aviation. As seen in Chapter 2 navigation primitives of current autopilots for unmanned vehicles are very simple and not very suitable for specifying complex missions. More advanced navigation systems, such as RNAV, can be found in commercial aviation. Nonetheless, RNAV has been designed with cargo and passengers transportation in mind and does not take into account the specific needs of the wide variety of UAS missions. Our proposed language borrows the leg concept from RNAV and extends it to accommodate iterative and conditional control constructs as well as other constructs, as the so-called parametric legs, that enable a simplified description of complex maneuvers. By reusing RNAV concepts we build on top of a solid base. In the long run, this choice could have the additional advantage of facilitating UAS integration in controlled airspace, since flight plans for both manned and unmanned systems would be specified in similar terms. The specification language is based on XML, a wide-spread technology that facilitates development of domain specific languages in a form that is both human readable and easy to process. An schema of the language has been developed in XSD, an XML schema definition language that can be used to guarantee syntactic correction of XML documents.

- The mechanisms for updating the flight plan and dynamically adapt the trajectory to the mission needs have also been defined. Utilization of iterative and conditional constructs already provides a level of dynamic adaptation that may suffice for some missions. However, in many cases, a more aggressive approach will be needed. For this reason the description of the UAS flight plan can be updated during mission time to better fit the mission needs. The commands for doing so also make use of XML in order to specify what needs to be changed. The possible content of these update messages is described in Chapter 4. The update mechanism enables not only changing leg parameters, such as the destination waypoint, but also adding or removing legs.

- Following the design explained in Chapter 5, a service called Flight Plan Manager has been implemented. The FPM is able to process and execute flight plans specified in our proposed language. To enable the service to operate with currently available autopilot systems, the flight plan primitives are dynamically translated to waypoints. The FPM is isolated from the specific details of each autopilot by virtue of the Virtual Autopilot System, which provides a standardized interface and handles the interaction with the installed autopilot. The VAS is one of the key services of the UAS architecture that the FPM integrates with. This architecture has been described in Chapter 3.

- The feasibility of the proposed approach for UAS flight plan management has been demonstrated by means of a simulated mission consisting in the flight inspection of Radio Navigation Aids. An experimental environment has been set up using FlightGear Flight Simulator to simulate the aircraft operation and Google Earth® for real time visualization of the mission evolution. This kind of mission will usually take place in controlled airspace. The simulation not only shows how the FPM is able to execute the mission with a high level of automation but also hints at possible ways of interaction between Air Traffic Controllers and the UAS operators. In particular, the UAS flight plan is divided into repeatable parts, with each one having a holding pattern assigned to it, whose location will have been agreed upon prior to the start of the mission. An ATC in continuous contact with the UAS operator may notify an interruption of the inspection at any time. The UAS operator will then command the UAS to fly the holding pattern assigned to its current procedures. As mentioned earlier, expressing the flight plan in ways ATCs are used to can facilitate UAS integration into controlled airspace.

- To integrate flight and payload operation, the Mission Manager service is added on top of the flight plan management capabilities. On the one hand, this mission management layer issues commands to mission related payload in order to adapt its operation to the current flight phase. On the other hand, it communicates with the FPM so that the UAS flight adapts to the mission needs. To enable support for a wide range of missions the specification of the UAS behavior is separated from the mechanisms used for its execution. In this way, the service responsible for mission execution becomes a general execution engine that carries out the submitted behavior specification. An aspect that differentiates our proposal from existing alternatives is the definition of the UAS flight and its mission related behavior into two separate documents. This approach enables a flight plan to be reused across different missions or even to operate without automated mission control. After a review of the state of the art on this topic, we propose SCXML as the language of choice for the task of specifying the UAS behavior. SCXML is a working draft from the World Wide Web Consortium (W3C) that provides a generic state-machine based execution environment based on Harel State Charts. In Chapter 7, an overview of the language features has been provided.

- As a proof of concept, a prototype of the Mission Manager has been developed. The implementation of the MMa is described in Chapter 7. The decision to use a future open standard has allowed us to take advantage of already existing tools such as Commons SCXML, an open source library that provides the SCXML execution engine. The MMa implements a wrapper around this library and integrates it into the distributed architecture described in Chapter 3.

- Finally, in Chapter 8 the MMa has been put to test in the context of a simulated hotspot detection mission. The mission has been carried out in two ways according to two different assumptions: (1) that our sensors and data processing services need some time to detect a potential hotspot and (2) that they are able to immediately detect a potential hotspot. This has lead to two different specifications of the UAS behavior where inspection of each hotspot is respectively done once the complete area under inspection has been scanned or at the very moment the hotspot is detected. In both cases, the same flight plan is used and only the mission specification differs. By performing the mission in these two different ways, the effectiveness of the proposed approach to provide a highly flexible and autonomous platform has been demonstrated.

9.2 Future Research

This section outlines several possible extensions of the work presented in this thesis.

A critical aspect that has not yet been addressed is how to guarantee flyability of all requested maneuvers. All flight plan procedures should be validated taking into account the aircraft performances and other factors such as weather conditions. This validation process could be performed on-ground before starting the mission, but changes to the flight plan can occur during flight time and strategies are needed for detecting non-flyable maneuvers and respond to such situation.

Another aspect not related to the aircraft capabilities that also requires validation of the flight plan consists in ensuring that no obstacles will be encountered. While this can easily be done during pre-flight for terrain and known fixed obstacles, again the ability of changing the flight plan during its execution forces us not only to be able to do this validation in a very efficient way during flight time, but also to come up with a rapid response to conflict situations.

The work presented in this dissertation enables the UAS to reactively respond to internal and external events while following pre-defined flight and mission plans. While we believe that this approach already results in a very capable platform the system could greatly benefit from the addition of deliberative capabilities. Flight planning capabilities could be used in the example mission of Chapter 8 to optimize the trajectory for visiting each one of the potential hotspots. In the example, when hotspot analysis is deferred, potential hotspots are visited in first in first out order, which clearly may not be the most efficient way for doing so.

Another area that needs to be explored relates to facilitating the task of flight plan and mission design. In this work, a new language has been developed for specifying flight plans and SCXML has been proposed for specifying UAS behavior. Encoding a complete mission using both languages is not trivial and additional research efforts should target at providing tools to facilitate this task. Many missions are similar in concept, for instance disaster damage assessment, search and rescue operations, crop monitoring and terrain mapping all involve a systematic observation of a given area as part of the mission. While each mission will have its own particularities, we believe that common patterns could be extracted and be used to provide means for designing a new mission by configuring different parameters of existing templates.

Along the lines of facilitating UAS mission design and operation, tools need to be developed for human operators to interact with the system. These tools should facilitate pre-flight tasks, enable supervision and control of the UAS during mission time and provide informative representations of what is going on to support decision making.

From a broader perspective, one of the most challenging problems faced by UAS, which involves both technical and non-technical issues, is their integration into non-segregated airspace. One of the contributions of this thesis has been the development of a flight plan specification language that can be used to describe the UAS flight path in ways that mimic current practices in commercial aviation. We believe that increasing convergence between the way manned and unmanned aircrafts operate will increase opportunities for these systems to share airspace. In that regard, the move that both the US (Cox *et al.* , 2009) and European (EUROCONTROL, 2009) Air Traffic Management Systems (ATMs) are experiencing towards new concepts of operation, which involve more reliance on digital communication infrastructures and automation, provides a great opportunity to research and define how, in the next years, UAS integration into controlled airspace should take place.

A XML Schemas

This appendix provides complete listings of XML schema definition documents for flight plan and flight plan updates. These documents can be used to validate syntactic correction of flight plans and their updates.

A.1 Flight Plan XML Schema

Listing A.1 provides complete specification of all valid elements and types that can be found in a flight plan specification.

Listing A.1: Flight plan XSD.

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://icarus.upc.es/schema/FlightPlan/1.1"
    xmlns:tns="http://icarus.upc.es/schema/FlightPlan/1.1"
    elementFormDefault="unqualified">
  <xsd:element name="FlightPlan">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Locale">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="speedUnits"
                  type="tns:SpeedUnitsEnum"/>
              <xsd:element name="angleUnits"
                  type="tns:AngleUnitsEnum"/>
              <xsd:element name="altitudeUnits"
                  type="tns:LengthUnitsEnum"/>
              <xsd:element name="distanceUnits"
                type="tns:LengthUnitsEnum"/>
              <xsd:element name="decimalSeparator"
                type="xsd:string"/>
              <xsd:element name="groupSeparator"
                type="xsd:string"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="Fixes" maxOccurs="1">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Fix" type="tns:FixType"
                maxOccurs="unbounded" minOccurs="0"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="EmergencyPlans">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="EmergencyFP"
                type="tns:FlightPlanType"
                maxOccurs="unbounded" minOccurs="0"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="MainFP" type="tns:FlightPlanType"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="FixType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="description" type="xsd:string"/>
      <xsd:element name="coordinates"
        type="xsd:string" minOccurs="1"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="tns:FixID"/>
</xsd:complexType>
<xsd:complexType name="StageType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"
      minOccurs="0"/>
    <xsd:element name="description" type="xsd:string"
      minOccurs="0"/>
    <xsd:element name="legs" minOccurs="0">
      <xsd:complexType>
        <xsd:sequence maxOccurs="unbounded">
          <xsd:element name="leg" type="tns:LegType"
            minOccurs="1"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="initialLegs" type="tns:LegIDREFS"
      minOccurs="0"/>
    <xsd:element name="finalLegs" type="tns:LegIDREFS"
      minOccurs="0"/>
    <xsd:element name="emergency" type="tns:FPIDREFS"
      minOccurs="0"/>
    <xsd:element name="groups" minOccurs="0">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="group">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="name"
                  type="xsd:string"/>
                <xsd:element name="description"
                  type="xsd:string" minOccurs="0"/>
                <xsd:element name="legList"
                  type="tns:LegIDREFS" minOccurs="0"/>
                <xsd:element name="groupList"
                  type="tns:GroupIDREFS" minOccurs="0"/>
              </xsd:sequence>
              <xsd:attribute name="id"
                type="tns:GroupID"/>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="targetId" type="xsd:string"
    use="optional"/>
  <xsd:attribute name="id" type="tns:StageID"
    use="optional"/>
  <xsd:attribute name="type" type="tns:StageEnumType"/>
  <xsd:attribute name="manualOnly" type="xsd:boolean"/>
</xsd:complexType>
<xsd:complexType name="LegType">
  <xsd:sequence>
    <xsd:element name="dest" type="tns:WaypointType"
      minOccurs="0"/>
    <xsd:element name="next" type="tns:LegIDREF"
      minOccurs="0"/>
    <xsd:element name="prev" type="tns:LegIDREF"
      minOccurs="0"/>
    <xsd:element name="emergency" type="tns:FPIDREF"
      minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="targetId" type="xsd:string"
    use="optional"/>
  <xsd:attribute name="id" type="tns:LegID"
    use="optional"/>
</xsd:complexType>
<xsd:complexType name="WaypointType">
  <xsd:sequence>
    <xsd:choice>
      <xsd:element name="fix" type="tns:FixIDREF"
        nillable="true" minOccurs="0"/>
      <xsd:element name="coordinates" type="xsd:string"
        minOccurs="0"/>
    </xsd:choice>
    <xsd:element name="fly-over" type="xsd:boolean"
      minOccurs="0"/>
    <xsd:element name="altitude" type="xsd:float"
      minOccurs="0"/>
    <xsd:element name="speed" type="xsd:float"
      minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:simpleType name="LegID">
  <xsd:restriction base="xsd:ID"/>
</xsd:simpleType>
<xsd:simpleType name="LegIDREF" id="LegID">
  <xsd:restriction base="xsd:IDREF"/>
</xsd:simpleType>
<xsd:simpleType name="FixID">
  <xsd:restriction base="xsd:ID"/>
</xsd:simpleType>
<xsd:simpleType name="FixIDREF" id="FixID">
  <xsd:restriction base="xsd:IDREF"/>
</xsd:simpleType>
<xsd:simpleType name="StageEnumType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Taxi"/>
    <xsd:enumeration value="TakeOff"/>
    <xsd:enumeration value="Departure"/>
    <xsd:enumeration value="Route"/>
    <xsd:enumeration value="Mission"/>
    <xsd:enumeration value="Arrival"/>
    <xsd:enumeration value="Approach"/>
    <xsd:enumeration value="Land"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="FlightPlanType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"
      minOccurs="0"/>
    <xsd:element name="description" type="xsd:string"
      minOccurs="0"/>
    <xsd:element name="stages" maxOccurs="1">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="stage"
            type="tns:StageType"
            maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="emergency"
      type="tns:FPIDREFS" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="targetId" type="xsd:string"
    use="optional"/>
  <xsd:attribute name="id" type="tns:FPID"
    use="optional"/>
  <xsd:attribute name="defaultTime" type="xsd:long"/>
  <xsd:attribute name="maxTime" type="xsd:long"/>
</xsd:complexType>
<xsd:simpleType name="FPID">
  <xsd:restriction base="xsd:ID"/>
</xsd:simpleType>
<xsd:simpleType name="FPIDREF">
  <xsd:restriction base="xsd:IDREF"/>
</xsd:simpleType>
<xsd:simpleType name="DirectionEnumType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="N">
    </xsd:enumeration>
    <xsd:enumeration value="S">
    </xsd:enumeration>
    <xsd:enumeration value="E">
    </xsd:enumeration>
    <xsd:enumeration value="W">
    </xsd:enumeration>
  </xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="IFLeg">
  <xsd:complexContent>
    <xsd:extension base="tns:LegType">
      <xsd:sequence/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="RFLeg">
  <xsd:complexContent>
    <xsd:extension base="tns:LegType">
      <xsd:sequence>
        <xsd:element name="center" type="xsd:string"/>
        <xsd:element name="direction"
          type="tns:TurnDirectionEnum"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:simpleType name="TurnDirectionEnum">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Left"/>
    <xsd:enumeration value="Right"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="TFLeg">
  <xsd:complexContent>
    <xsd:extension base="tns:LegType">
      <xsd:sequence/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="IterativeLeg">
  <xsd:complexContent>
    <xsd:extension base="tns:LegType">
      <xsd:sequence>
        <xsd:element name="body" type="tns:LegIDREFS"/>
        <xsd:element name="first" type="tns:LegIDREF"/>
        <xsd:element name="last" type="tns:LegIDREF"/>
        <xsd:element name="upperBound" type="xsd:int"
          minOccurs="0"/>
        <xsd:element name="cond" type="xsd:string"
          minOccurs="0"/>
      </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:simpleType name="SpeedUnitsEnum">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="ms">
      </xsd:enumeration>
      <xsd:enumeration value="kt">
      </xsd:enumeration>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="AngleUnitsEnum">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="deg">
      </xsd:enumeration>
      <xsd:enumeration value="rad">
      </xsd:enumeration>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="LengthUnitsEnum">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="m">
      </xsd:enumeration>
      <xsd:enumeration value="nm">
      </xsd:enumeration>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="IntersectionLeg">
    <xsd:complexContent>
      <xsd:extension base="tns:LegType">
        <xsd:sequence>
          <xsd:element name="nextCond" type="xsd:string"
            minOccurs="0"/>
          <xsd:element name="nextList" type="tns:LegIDREFS"
            minOccurs="0"/>
          <xsd:element name="prevCond" type="xsd:string"
            minOccurs="0"/>
          <xsd:element name="prevList" type="tns:LegIDREFS"
            minOccurs="0"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:simpleType name="StageID">
    <xsd:restriction base="xsd:ID"/>
  </xsd:simpleType>
  <xsd:complexType name="HFLeg">
    <xsd:complexContent>
      <xsd:extension base="tns:LegType">
        <xsd:sequence>
          <xsd:element name="course" type="xsd:double"
            minOccurs="0"/>
          <xsd:element name="direction"
            type="tns:TurnDirectionEnum" minOccurs="0"/>
          <xsd:element name="d1" type="xsd:double"
            minOccurs="0"/>
          <xsd:element name="d2" type="xsd:double"
            minOccurs="0"/>
          <xsd:element name="upperBound" type="xsd:int"
            minOccurs="0"/>
          <xsd:element name="cond" type="xsd:string"
            minOccurs="0"/>
          <xsd:element name="altitude" type="xsd:double"
            minOccurs="0"/>
          <xsd:element name="climbRate" type="xsd:double"
            minOccurs="0"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="BasicScanLeg">
    <xsd:complexContent>
      <xsd:extension base="tns:LegType">
        <xsd:sequence>
          <xsd:element name="startAt" type="xsd:string"
            minOccurs="0"/>
          <xsd:element name="dim1" type="xsd:double"
            minOccurs="0"/>
          <xsd:element name="dim2" type="xsd:double"
            minOccurs="0"/>
          <xsd:element name="angle" type="xsd:double"
            minOccurs="0"/>
          <xsd:element name="separation" type="xsd:double"
            minOccurs="0"/>
          <xsd:element name="turndirection"
            type="tns:TurnDirectionEnum" minOccurs="0"/>
          <xsd:element name="d1" type="xsd:double"
            minOccurs="0"/>
          <xsd:element name="d2" type="xsd:double"
            minOccurs="0"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:simpleType name="LegIDREFS">
    <xsd:restriction base="xsd:IDREFS"/>
  </xsd:simpleType>
  <xsd:attribute name="syntax" type="xsd:string"/>
  <xsd:complexType name="RwyFixType">
    <xsd:complexContent>
      <xsd:extension base="tns:FixType">
        <xsd:sequence>
          <xsd:element name="altitude" type="xsd:double"/>
          <xsd:element name="heading" type="xsd:double"/>
          <xsd:element name="length" type="xsd:double"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:simpleType name="GroupID">
    <xsd:restriction base="xsd:ID"/>
  </xsd:simpleType>
  <xsd:simpleType name="GroupIDREFS">
    <xsd:restriction base="xsd:IDREFS"/>
  </xsd:simpleType>
  <xsd:complexType name="DFLeg">
    <xsd:complexContent>
      <xsd:extension base="tns:LegType">
        <xsd:sequence/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="ScanPointLeg">
    <xsd:complexContent>
      <xsd:extension base="tns:LegType">
        <xsd:sequence>
          <xsd:element name="course" type="xsd:double"
            minOccurs="0"/>
          <xsd:element name="d1" type="xsd:double"
            minOccurs="0"/>
          <xsd:element name="d2" type="xsd:double"
            minOccurs="0"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:simpleType name="FPIDREFS">
    <xsd:restriction base="tns:FPIDREF"
  xmlns:tns="http://icarus.upc.es/schema/FlightPlan/1.1"/>
  </xsd:simpleType>
</xsd:schema>
```

A.2 FP Updates XML Schema

All flight plan updates must conform to the schema definition provided in Listing A.2. This specification imports and reuses elements from the flight plan schema.

Listing A.2: Flight plan updates XSD.

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:fps="http://icarus.upc.es/schema/FlightPlan/1.1"
  targetNamespace="http://icarus.upc.es/schema/FlightPlanUpdate/1.1"
  xmlns:tns="http://icarus.upc.es/schema/FlightPlanUpdate/1.1"
  elementFormDefault="unqualified">

  <xsd:import namespace="http://icarus.upc.es/schema/FlightPlan/1.1"
    schemaLocation="FlightPlan-1.1.xsd"/>

  <xsd:element name="FlightPlanUpdate">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Add"
          minOccurs="0">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Fixes"
                minOccurs="0">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="Fix"
                      type="fps:FixType"
                    maxOccurs="unbounded"/>
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
            <xsd:element name="EmergencyPlans"
              minOccurs="0">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="EmergencyFp"
                    type="fps:FlightPlanType"
                    maxOccurs="unbounded"/>
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
            <xsd:element name="MainFP"
              type="fps:FlightPlanType"
              minOccurs="0"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="Change"
        minOccurs="0">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="EmergencyPlans"
              minOccurs="0">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="EmergencyFp"
                    type="fps:FlightPlanType"
                    maxOccurs="unbounded"/>
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
            <xsd:element name="MainFP"
              type="fps:FlightPlanType"
              minOccurs="0"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="Delete"
        minOccurs="0">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="EmergencyPlans"
              minOccurs="0">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="EmergencyFp"
                    type="fps:FlightPlanType"
                    maxOccurs="unbounded"/>
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
            <xsd:element name="MainFP"
              type="fps:FlightPlanType"
              minOccurs="0"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

B
Specification of Navaids Inspection Mission

Listing B.1 provides the specification of the flight plan for the navaid flight inspection mission. In this flight plan all inspection procedures are broken down into small legs. All this legs are put together inside an iterative construct so that we can always go back and repeat part of the flight. Smaller legs provide finer grain control over parts that should be repeated. In this mission we only make use of the Flight Plan Manager, the Mission Manager is not involved, therefore, only the flight plan specification is needed.

Listing B.1: XML flight plan.

```
<?xml version="1.0" encoding="UTF-8"?>

<fp:FlightPlan
   xmlns:fp='http://icarus.upc.es/schema/FlightPlan/1.1'
   xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
   xsi:schemaLocation='file:///FlightPlan-1.1.xsd'>

  <!-- Locale settings -->
  <Locale>
    <speedUnits>kt</speedUnits>
    <altitudeUnits>m</altitudeUnits>
    <distanceUnits>m</distanceUnits>
    <decimalSeparator>.</decimalSeparator>
    <groupSeparator/>
  </Locale>

  <!-- List of fixes -->
  <Fixes>
    <Fix id="LEHC12R" xsi:type="fp:RwyFixType">
      <name>Huesca-Pirineos LEHC 12R Runway</name>
      <description>Runway 12 Huesca Airport</description>
              <coordinates>42.0809 -0.3273</coordinates>
      <altitude>539</altitude>
      <heading>124</heading>
      <length>2100</length>
    </Fix>
  </Fixes>

  <!-- List of emergency flight plans -->
  <EmergencyPlans/>

  <!-- Main flight plan -->
  <MainFP id="VOR-HUE">
    <name>A periodic inspetion of VOR-HUE</name>
    <description>A periodic inspection of VOR-HUE</description>
    <stages>
      <stage id="taxi" manualOnly="true" type="Taxi"/>

      <stage id="takeoff" manualOnly="true" type="TakeOff"/>

      <stage id="depart" type="Departure">
        <name>Departure Procedure</name>
        <description>Go to route start</description>
        <legs>
          <leg id="depart1" xsi:type="fp:TFLeg">
            <dest>
              <coordinates>41.6785 -0.9677</coordinates>
              <altitude>1000</altitude>
              <speed>160</speed>
            </dest>
          </leg>
        </legs>
        <initialLegs>depart1</initialLegs>
        <finalLegs>depart1</finalLegs>
      </stage>

      <stage id="goroute" type="Route">
        <name>To Mission Route</name>
        <description>Go to mission area</description>
    <legs>
      <leg id="VOR-VOID-0-A" xsi:type="fp:TFLeg">
        <dest>
          <coordinates>41.7992 -0.6844</coordinates>
          <fly-over>true</fly-over>
          <altitude>2000</altitude>
          <speed>160</speed>
        </dest>
      </leg>
    </legs>
    <initialLegs>VOR-VOID-0-A</initialLegs>
    <finalLegs>VOR-VOID-0-A</finalLegs>
  </stage>

  <stage id="mission" type="Mission">
    <name>Mission</name>
    <description>Perform mission</description>
    <legs>
      <leg id="Loop" xsi:type="fp:IterativeLeg">
        <body>
            Cond
            Holding
            VOR-REF-1-A
            VOR-REF-1-B
            VOR-VOID-1-A
            VOR-VOID-1-B
            VOR-VOID-1-C
            VOR-REF-2-A
            VOR-REF-2-B
            VOR-VOID-2-A
            VOR-ORB360R10-1-A
            VOR-ORB360R10-1-B
            VOR-ORB360R10-1-C
            VOR-ORB360R10-1-D
            VOR-ORB360R10-2-A
            VOR-ORB360R10-2-B
            VOR-ORB360R10-2-C
            VOR-ORB360R10-2-D
            VOR-VOID-3-A
            VOR-VOID-3-B
            VOR-RAD-1-A
            VOR-APP-1-A
            VOR-APP-1-B
            VOR-APP-1-C
            VOR-APP-1-D
            VOR-SIDSTAR-1-A
            VOR-SIDSTAR-1-B
            VOR-SIDSTAR-1-C
            Join
        </body>
        <first>Cond</first>
        <last>Join</last>
        <upperBound>1</upperBound>
        <cond>BreakIteration</cond>
      </leg>
      <leg id="Cond" xsi:type="fp:IntersectionLeg">
        <next>VOR-REF-1-A</next>
        <nextCond>Holding</nextCond>
        <nextList>VOR-REF-1-A Holding</nextList>
      </leg>
      <leg id="Holding" xsi:type="fp:HFLeg">
        <dest>
          <coordinates>41.8089 -0.5276</coordinates>
          <altitude>2000</altitude>
          <speed>160</speed>
        </dest>
        <next>Join</next>
        <course>30</course>
        <direction>Right</direction>
        <d1>5000</d1>
        <d2>3000</d2>
        <upperBound>5</upperBound>
        <cond>BreakHolding</cond>
      </leg>
      <leg id="VOR-REF-1-A" xsi:type="fp:TFLeg">
        <dest>
          <coordinates>41.8181 -0.6599</coordinates>
          <fly-over>true</fly-over>
          <altitude>2000</altitude>
          <speed>200</speed>
        </dest>
        <next>VOR-REF-1-B</next>
      </leg>
      <leg id="VOR-REF-1-B" xsi:type="fp:TFLeg">
        <dest>
          <coordinates>42.0733 -0.3189</coordinates>
          <fly-over>true</fly-over>
          <altitude>2000</altitude>
          <speed>200</speed>
        </dest>
        <next>VOR-VOID-1-A</next>
      </leg>
      <leg id="VOR-VOID-1-A" xsi:type="fp:DFLeg">
        <dest>
          <coordinates>42.0924 -0.2932</coordinates>
          <altitude>2000</altitude>
          <speed>160</speed>
        </dest>
        <next>VOR-VOID-1-B</next>
      </leg>
      <leg id="VOR-VOID-1-B" xsi:type="fp:DFLeg">
        <dest>
          <coordinates>42.0776 -0.2553</coordinates>
          <fly-over>true</fly-over>
          <altitude>2000</altitude>
          <speed>160</speed>
        </dest>
        <next>VOR-VOID-1-C</next>
      </leg>
      <leg id="VOR-VOID-1-C" xsi:type="fp:TFLeg">
        <dest>
          <coordinates>42.0580 -0.2794</coordinates>
          <fly-over>true</fly-over>
          <altitude>2000</altitude>
          <speed>160</speed>
        </dest>
        <next>VOR-REF-2-A</next>
      </leg>
      <leg id="VOR-REF-2-A" xsi:type="fp:TFLeg">
        <dest>
          <coordinates>42.0733 -0.3189</coordinates>
          <fly-over>true</fly-over>
          <altitude>2000</altitude>
          <speed>200</speed>
        </dest>
        <next>VOR-REF-2-B</next>
      </leg>
      <leg id="VOR-REF-2-B" xsi:type="fp:TFLeg">
        <dest>
          <coordinates>42.2821 -0.6684</coordinates>
          <fly-over>true</fly-over>
          <altitude>2000</altitude>
          <speed>200</speed>
        </dest>
        <next>VOR-VOID-2-A</next>
      </leg>
      <leg id="VOR-VOID-2-A" xsi:type="fp:DFLeg">
        <dest>
          <coordinates>42.2173 -0.4318</coordinates>
          <altitude>1470</altitude>
          <speed>200</speed>
        </dest>
        <next>VOR-ORB360R10-1-A</next>
      </leg>
      <leg id="VOR-ORB360R10-1-A" xsi:type="fp:RFLeg">
        <dest>
          <coordinates>42.1572 -0.1254</coordinates>
          <altitude>2000</altitude>
          <speed>200</speed>
        </dest>
        <next>VOR-ORB360R10-1-B</next>
        <center>42.0733 -0.3189</center>
        <direction>Right</direction>
      </leg>
      <leg id="VOR-ORB360R10-1-B" xsi:type="fp:RFLeg">
        <dest>
          <coordinates>41.9276 -0.2104</coordinates>
          <altitude>2000</altitude>
          <speed>200</speed>
        </dest>
        <next>VOR-ORB360R10-1-C</next>
        <center>42.0733 -0.3189</center>
        <direction>Right</direction>
      </leg>
      <leg id="VOR-ORB360R10-1-C" xsi:type="fp:RFLeg">
        <dest>
          <coordinates>41.9891 -0.5119</coordinates>
          <altitude>2000</altitude>
          <speed>200</speed>
        </dest>
        <next>VOR-ORB360R10-1-D</next>
        <center>42.0733 -0.3189</center>
        <direction>Right</direction>
      </leg>
      <leg id="VOR-ORB360R10-1-D" xsi:type="fp:RFLeg">
        <dest>
          <coordinates>42.2235 -0.4162</coordinates>
          <altitude>2000</altitude>
          <speed>200</speed>
        </dest>
        <next>VOR-ORB360R10-2-A</next>
        <center>42.0733 -0.3189</center>
        <direction>Right</direction>
      </leg>
      <leg id="VOR-ORB360R10-2-A" xsi:type="fp:RFLeg">
        <dest>
          <coordinates>42.1572 -0.1254</coordinates>
          <altitude>2000</altitude>
          <speed>200</speed>
        </dest>
        <next>VOR-ORB360R10-2-B</next>
        <center>42.0733 -0.3189</center>
        <direction>Right</direction>
      </leg>
      <leg id="VOR-ORB360R10-2-B" xsi:type="fp:RFLeg">
        <dest>
          <coordinates>41.9276 -0.2104</coordinates>
          <altitude>2000</altitude>
      <speed>200</speed>
    </dest>
    <next>VOR-ORB360R10-2-C</next>
    <center>42.0733 -0.3189</center>
    <direction>Right</direction>
  </leg>
  <leg id="VOR-ORB360R10-2-C" xsi:type="fp:RFLeg">
    <dest>
      <coordinates>41.9891 -0.5119</coordinates>
      <altitude>2000</altitude>
      <speed>200</speed>
    </dest>
    <next>VOR-ORB360R10-2-D</next>
    <center>42.0733 -0.3189</center>
    <direction>Right</direction>
  </leg>
  <leg id="VOR-ORB360R10-2-D" xsi:type="fp:RFLeg">
    <dest>
      <coordinates>42.2379 -0.3543</coordinates>
      <altitude>2000</altitude>
      <speed>200</speed>
    </dest>
    <next>VOR-VOID-3-A</next>
    <center>42.0733 -0.3189</center>
    <direction>Right</direction>
  </leg>
  <leg id="VOR-VOID-3-A" xsi:type="fp:DFLeg">
    <dest>
      <coordinates>42.0498 0.0125</coordinates>
    </dest>
    <next>VOR-VOID-3-B</next>
  </leg>
  <leg id="VOR-VOID-3-B" xsi:type="fp:TFLeg">
    <dest>
      <coordinates>42.0614 -0.1572</coordinates>
      <fly-over>true</fly-over>
      <altitude>1410</altitude>
      <speed>160</speed>
    </dest>
    <next>VOR-RAD-1-A</next>
  </leg>
  <leg id="VOR-RAD-1-A" xsi:type="fp:TFLeg">
    <dest>
      <coordinates>42.0733 -0.3189</coordinates>
      <fly-over>true</fly-over>
    </dest>
    <next>VOR-VOID-4-A</next>
  </leg>
  <leg id="VOR-VOID-4-A" xsi:type="fp:DFLeg">
    <dest>
      <coordinates>42.0799 -0.4470</coordinates>
      <altitude>1486</altitude>
      <speed>160</speed>
    </dest>
    <next>VOR-VOID-4-B</next>
  </leg>
  <leg id="VOR-VOID-4-B" xsi:type="fp:DFLeg">
    <dest>
      <coordinates>42.1336 -0.4056</coordinates>
    </dest>
    <next>VOR-VOID-4-C</next>
  </leg>
  <leg id="VOR-VOID-4-C" xsi:type="fp:TFLeg">
    <dest>
      <coordinates>42.0733 -0.3189</coordinates>
      <fly-over>true</fly-over>
    </dest>
    <next>VOR-APP-1-A</next>
  </leg>
  <leg id="VOR-APP-1-A" xsi:type="fp:TFLeg">
    <dest>
      <coordinates>41.9478 -0.1718</coordinates>
      <fly-over>true</fly-over>
      <altitude>1189</altitude>
      <speed>200</speed>
    </dest>
    <next>VOR-APP-1-B</next>
  </leg>
  <leg id="VOR-APP-1-B" xsi:type="fp:RFLeg">
    <dest>
      <coordinates>41.9878 -0.1270</coordinates>
      <altitude>1158</altitude>
      <speed>160</speed>
    </dest>
    <next>VOR-APP-1-C</next>
    <center>41.9652 -0.1451</center>
    <direction>Left</direction>
  </leg>
  <leg id="VOR-APP-1-C" xsi:type="fp:TFLeg">
    <dest>
      <coordinates>42.0306 -0.2229</coordinates>
      <fly-over>true</fly-over>
      <altitude>975</altitude>
      <speed>140</speed>
    </dest>
    <next>VOR-APP-1-D</next>
  </leg>
  <leg id="VOR-APP-1-D" xsi:type="fp:TFLeg">
    <dest>
```

```
        <coordinates>42.0733 -0.3189</coordinates>
        <fly-over>true</fly-over>
        <altitude>1036</altitude>
        <speed>140</speed>
      </dest>
      <next>VOR-SIDSTAR-1-A</next>
    </leg>
    <leg id="VOR-SIDSTAR-1-A" xsi:type="fp:TFLeg">
      <dest>
        <coordinates>42.1514 -0.4396</coordinates>
        <altitude>1310</altitude>
        <speed>160</speed>
      </dest>
      <next>VOR-SIDSTAR-1-B</next>
    </leg>
    <leg id="VOR-SIDSTAR-1-B" xsi:type="fp:DFLeg">
      <dest>
        <coordinates>42.0714 -0.4828</coordinates>
        <altitude>1410</altitude>
        <speed>200</speed>
      </dest>
      <next>VOR-SIDSTAR-1-C</next>
    </leg>
    <leg id="VOR-SIDSTAR-1-C" xsi:type="fp:TFLeg">
      <dest>
        <coordinates>42.0733 -0.3189</coordinates>
      </dest>
      <next>Join</next>
    </leg>
    <leg id="Join" xsi:type="fp:IntersectionLeg"/>
  </legs>
  <initialLegs>Loop</initialLegs>
  <finalLegs>Loop</finalLegs>
</stage>

<stage id="retroute" type="Route">
  <name>Return Route</name>
  <description>Return from mission area</description>
  <legs>
    <leg id="VOR-FINISH-1-A" xsi:type="fp:TFLeg">
      <dest>
        <coordinates>41.9478 -0.1718</coordinates>
        <fly-over>true</fly-over>
        <altitude>1189</altitude>
        <speed>200</speed>
      </dest>
    </leg>
  </legs>
  <initialLegs>VOR-FINISH-1-A</initialLegs>
  <finalLegs>VOR-FINISH-1-A</finalLegs>
</stage>

<stage id="arrival" type="Arrival">
  <name>Arrival stage</name>
  <description></description>
  <legs>
    <leg id="VOR-ARR-1-A" xsi:type="fp:RFLeg">
      <dest>
        <coordinates>41.9878 -0.1270</coordinates>
        <altitude>1158</altitude>
        <speed>160</speed>
      </dest>
      <center>41.9652 -0.1451</center>
      <direction>Left</direction>
    </leg>
  </legs>
  <initialLegs>VOR-ARR-1-A</initialLegs>
  <finalLegs>VOR-ARR-1-A</finalLegs>
</stage>

<stage id="approach" type="Approach">
  <name>Approach</name>
  <description></description>
  <legs>
    <leg id="VOR-APP-A" xsi:type="fp:TFLeg">
      <dest>
        <coordinates>42.0306 -0.2229</coordinates>
        <fly-over>true</fly-over>
        <altitude>975</altitude>
        <speed>140</speed>
      </dest>
      <next>VOR-APP-B</next>
    </leg>
    <leg id="VOR-APP-B" xsi:type="fp:TFLeg">
      <dest>
        <coordinates>42.0733 -0.3189</coordinates>
        <fly-over>true</fly-over>
        <altitude>1036</altitude>
        <speed>140</speed>
      </dest>
    </leg>
  </legs>
  <initialLegs>VOR-APP-A</initialLegs>
  <finalLegs>VOR-APP-B</finalLegs>
</stage>

<stage id="land" manualOnly="true" type="Land"/>

<stage id="taxi2" manualOnly="true" type="Taxi"/>
```

```
  </stages>
</MainFP>
</fp:FlightPlan>
```

C Specification of Hotspot Detection Mission

This appendix provides complete listings of flight plan and mission specifications used in the simulation of the hotspot detection mission.

C.1 Flight Plan Specification

Listing C.1 provides the specification of the flight plan for carrying out the hotspot detection mission. The same flight plan is used in the two strategies used for performing the mission.

Listing C.1: XML flight plan specification.

```
<?xml version="1.0" encoding="UTF-8"?>

<fp:FlightPlan
  xmlns:fp='http://icarus.upc.es/schema/FlightPlan/1.1'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:schemaLocation='file:FlightPlan-1.1.xsd'>

  <!-- Locale settings -->
  <Locale>
    <speedUnits>kt</speedUnits>
    <altitudeUnits>m</altitudeUnits>
    <distanceUnits>m</distanceUnits>
    <decimalSeparator>.</decimalSeparator>
    <groupSeparator />
  </Locale>

  <!-- List of fixes -->
  <Fixes>
    <Fix id="LEIG17" xsi:type="fp:RwyFixType">
      <name>Igualada-Odena LEIG 17 Runway</name>
      <description>
        Runway 17 of Igualada-Odena airport
      </description>
      <coordinates>
        41.58680712601129 1.651446229162748
      </coordinates>
      <altitude>331</altitude>
      <heading>170</heading>
      <length>780</length>
    </Fix>
    <Fix id="LEIG35" xsi:type="fp:RwyFixType">
      <name>Igualada-Odena LEIG 35 Runway</name>
      <description>
        Runway 35 of Igualada-Odena airport
      </description>
      <coordinates>
        41.58098991505337 1.654476003345295
      </coordinates>
      <altitude>321</altitude>
      <heading>310</heading>
      <length>780</length>
    </Fix>
  </Fixes>

  <!-- List of emergency flight plans -->
  <EmergencyPlans />

  <!-- Main flight plan -->
```

```xml
<MainFP id="HotSpotPlan">
  <name>Hot Spot Mission Flight Plan</name>
  <description>
    Perform hot spot mission
  </description>
  <stages>
    <!-- Taxi to runway -->
    <stage id="taxi" manualOnly="true" type="Taxi"/>

    <!-- Take Off from rwy LEIG17 -->
    <stage id="takeoff" manualOnly="true" type="TakeOff"/>

    <!-- Depart stage -->
    <stage id="depart" type="Departure">
      <name>Departure Procedure</name>
      <description>
        Go to route start
      </description>
      <legs>
        <leg id="departLeg" xsi:type="fp:TFLeg">
          <dest>
            <coordinates>
              41.57952426103741 1.670437018720817
            </coordinates>
            <altitude>550</altitude>
            <speed>70</speed>
          </dest>
        </leg>
      </legs>
      <initialLegs>departLeg</initialLegs>
      <finalLegs>departLeg</finalLegs>
    </stage>

    <!-- Go to mission area -->
    <stage id="goroute" type="Route">
      <name>To Mission Route</name>
      <description>Go to mission area</description>
      <legs>
        <leg id="rleg" xsi:type="fp:TFLeg">
          <dest>
            <coordinates>
              41.58472121226085 1.684652022474151
            </coordinates>
            <altitude>800</altitude>
          </dest>
        </leg>
      </legs>
      <initialLegs>rleg</initialLegs>
      <finalLegs>rleg</finalLegs>
    </stage>

    <!-- Perform mission -->
    <stage id="mission" type="Mission">
      <name>Scan area mission</name>
      <description>Scan over the area</description>
      <legs>
        <leg id="missloop" xsi:type="fp:IterativeLeg">
          <body>
            patternSelect scanArea scanPoint hold join
          </body>
          <first>patternSelect</first>
          <last>join</last>
          <upperBound>15</upperBound>
        </leg>
        <leg id="patternSelect"
          xsi:type="fp:IntersectionLeg">
          <next>scanArea</next> <!-- default value -->
          <nextCond>selection</nextCond>
          <nextList>scanArea scanPoint hold</nextList>
        </leg>
        <leg id="scanArea" xsi:type="fp:BasicScanLeg">
          <dest>
            <coordinates>
              41.5493424917977 1.77254310685181
            </coordinates>
            <speed>60</speed>
          </dest>
          <next>join</next>
          <dim1>6000</dim1>
          <dim2>5500</dim2>
          <angle>80</angle>
          <separation>800</separation>
        </leg>
        <leg id="scanPoint" xsi:type="fp:ScanPointLeg">
          <dest>
            <coordinates>
              41.56947331267459 1.717810982215079
            </coordinates>
            <fly-over>true</fly-over>
            <speed>40</speed>
          </dest>
          <next>join</next>
          <course>135</course>
          <d1>1000</d1>
          <d2>450</d2>
        </leg>
        <leg id="hold" xsi:type="fp:HFLeg">
          <dest>
            <coordinates>
              41.55523585866938 1.777892046315137
            </coordinates>
            <speed>40</speed>
          </dest>
          <next>join</next>
          <course>45</course>
          <direction>Right</direction>
          <d1>1000</d1>
          <d2>450</d2>
        </leg>
        <leg id="join" xsi:type="fp:IntersectionLeg" />
      </legs>
      <initialLegs>missloop</initialLegs>
      <finalLegs>missloop</finalLegs>
    </stage>

    <!-- Return from mission area route -->
    <stage id="retroute" type="Route">
      <name>Return Route</name>
      <description>Return from mission</description>
      <legs>
        <leg id="retleg" xsi:type="fp:TFLeg">
          <dest>
            <coordinates>
              41.56510942654809 1.686081825185838
            </coordinates>
            <speed>70</speed>
          </dest>
        </leg>
      </legs>
      <initialLegs>retleg</initialLegs>
      <finalLegs>retleg</finalLegs>
    </stage>

    <!-- Arrival stage -->
    <stage id="arrival" type="Arrival">
      <name>Arrival stage</name>
      <description />
      <legs>
        <leg id="arrileg" xsi:type="fp:TFLeg">
          <dest>
            <coordinates>
              41.5748709413011 1.667392541543762
            </coordinates>
          </dest>
        </leg>
      </legs>
      <initialLegs>arrileg</initialLegs>
      <finalLegs>arrileg</finalLegs>
    </stage>

    <!-- Approach stage -->
    <stage id="approach" type="Approach">
      <name>Approach</name>
      <description />
      <legs>
        <leg id="appr" xsi:type="fp:TFLeg">
          <dest>
            <coordinates>
              41.57813348412295 1.656011088093122
            </coordinates>
            <altitude>50</altitude>
            <speed>15</speed>
          </dest>
        </leg>
      </legs>
      <initialLegs>appr</initialLegs>
      <finalLegs>appr</finalLegs>
    </stage>

    <!-- Landing stage -->
    <stage id="land" manualOnly="true" type="Land"/>

    <stage id="taxi2" manualOnly="true" type="Taxi"/>
  </stages>
</MainFP>
</fp:FlightPlan>
```

C.2 Hotspot Mission Main States

Listing C.2 defines the main states of the mission in SCXML. The *src* attribute in the *Mission* state is used to select the file that contains the specification of all the mission details.

Listing C.2: SCXML main states specification.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
   Definition of UAS behavior for HotSpot mission.
-->
<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=='retroute'" target="EnRoute" />
  </state>

  <state id="Landing">
    <transition event="on_ground" target="OnGround" />
  </state>

</scxml>
```

C.3 Deferred Hotspot Analysis

Listing C.3 specifies the behavior of the UAS during the mission state when applying a deferred hotspot analysis strategy. Two parallel states are respectively in charge of counting the number of potential hotspots and managing the UAS flight. Several *datamodel* elements are used to hold variables and templates of the update messages that will be sent to the FPM. This data is accessed in various expressions and modified using the *assign* SCXML element. *Send* is used to notify the MMa service, that hosts the execution engine, that an outbound event has occurred.

Listing C.3: SCXML deferred analysis specification.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
   Definition of UAS behavior for HotSpot mission.
-->
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0"
   initialstate="HotSpotProcessing">

  <datamodel>
    <data id="selection" expr="0" />
   <!-- Detected HotSpots Counter -->
    <data id="detect_hs_count" expr="0" />
   <!-- Visited HotSpots Counter -->
    <data id="visit_hs_count" expr="0" />
  </datamodel>

  <parallel id="HotSpotProcessing">

    <!-- Keep track of the number of potential hotspots -->
    <state id="HotSpotsCounter">
      <datamodel>
        <data id="scanPointUpdate"
         xmlns:fpu='http://icarus.upc.es/schema/FlightPlanUpdate/1.1'
         xmlns:fp='http://icarus.upc.es/schema/FlightPlan/1.1'
         xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'>
          <fpu:FlightPlanUpdate xmlns="">
            <Change>
              <MainFP targetId="HotSpotPlan">
                <stages>
                  <stage targetId="mission">
                    <legs>
                      <leg targetId="scanPoint" xsi:type="fp:ScanPointLeg">
                        <dest>
                          <coordinates />
                        </dest>
                      </leg>
                </legs>
              </stage>
            </stages>
          </MainFP>
        </Change>
      </fpu:FlightPlanUpdate>
    </data>
  <!-- Auxiliary variables -->
    <data id="lat" expr="" />
    <data id="lon" expr="" />
    <data id="coordinates" expr="" />
  </datamodel>

  <transition event="hotspot">
    <assign name="detect_hs_count" expr="detect_hs_count + 1" />
    <log expr="'Number of detected hot spots: ' + detect_hs_count" />

  <!-- Set scanPoint target to first non-visited potential hotspot -->
    <assign name="lat" expr="HotSpotList[visit_hs_count].getLatitude().toString()" />
    <assign name="lon" expr="HotSpotList[visit_hs_count].getLongitude().toString()" />
    <assign name="coordinates" expr="lat.concat(' ').concat(lon)" />
    <log expr="'HotSpot location: ' + coordinates"/>
    <assign
       xmlns:fpu='http://icarus.upc.es/schema/FlightPlanUpdate/1.1'
       location="Data(scanPointUpdate,
       'fpu:FlightPlanUpdate/Change/MainFP/stages/stage/legs/leg/dest/coordinates')"
       expr="coordinates" />
    <!-- Send message to fpms -->
    <send target="'container'" targettype="'x-marea'" event="'update_cmd'" namelist="scanPointUpdate" />
    <!-- Set scanPoint as next leg -->
    <assign name="selection" expr="1" />
    <send target="'container'" targettype="'x-marea'" event="'set_condition'" namelist="selection" />
  </transition>
</state>

<!-- Perform mission -->
<state id="HotSpotAnalysis" >
  <initial>
    <transition target="ScanArea"/>
  </initial>

  <!-- During the mission stage, the UAS will be in one of the following states:
          Hold: Don't know what to do, just wait.
          Scan Area: Scan the area of interest.
          Scan Point: Take closer look at a potential hot spot. -->
  <state id="ScanArea">
    <!-- Scan area -->
    <onentry>
      <!-- When done go to hold state -->
      <assign name="selection" expr="2" />
      <send target="'container'" targettype="'x-marea'" event="'set_condition'" namelist="selection" />
    </onentry>

    <transition event="current_leg" cond="_eventdata=='hold'" target="Hold" />
    <transition event="current_leg" cond="_eventdata=='scanPoint'" target="ScanPoint" />
  </state>

  <state id="Hold">
    <transition event="current_leg" cond="_eventdata=='scanPoint'" target="ScanPoint" />
    <transition event="scan" target="ScanArea" />
  </state>

  <!-- Take closer look at hot spot -->
  <state id="ScanPoint">
    <datamodel>
      <!-- Update message for scan point -->
      <data id="scanPointUpdate"
         xmlns:fpu='http://icarus.upc.es/schema/FlightPlanUpdate/1.1'
         xmlns:fp='http://icarus.upc.es/schema/FlightPlan/1.1'
         xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'>
        <fpu:FlightPlanUpdate xmlns="">
          <Change>
            <MainFP targetId="HotSpotPlan">
              <stages>
                <stage targetId="mission">
                  <legs>
                    <leg targetId="scanPoint" xsi:type="fp:ScanPointLeg">
                      <dest>
                        <coordinates />
                      </dest>
                    </leg>
                  </legs>
                </stage>
              </stages>
            </MainFP>
          </Change>
        </fpu:FlightPlanUpdate >
      </data>
      <!-- Auxiliary variables -->
      <data id="lat" expr="" />
      <data id="lon" expr="" />
      <data id="coordinates" expr="" />
    </datamodel>

    <onentry>
      <assign name="visit_hs_count" expr="visit_hs_count + 1" />
      <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()" />
```

```
            <assign name="lon" expr="HotSpotList[visit_hs_count].getLongitude().toString()" />
            <assign name="coordinates" expr="lat.concat(' ').concat(lon)" />
            <log expr="'HotSpot location: ' + coordinates"/>
            <assign
              xmlns:fpu='http://icarus.upc.es/schema/FlightPlanUpdate/1.1'
              location="Data(scanPointUpdate,
                     'fpu:FlightPlanUpdate/Change/MainFP/stages/stage/legs/leg/dest/coordinates')"
              expr="coordinates" />
            <!-- Send message to fpms -->
            <send target="'container'" targettype="'x-marea'" event="'update_cmd'" namelist="scanPointUpdate" />
            <!-- Set scanPoint as next leg -->
            <assign name="selection" expr="1" />
            <send target="'container'" targettype="'x-marea'" event="'set_condition'" namelist="selection" />
          <else/>
            <!-- When done go to hold -->
            <assign name="selection" expr="2" />
            <send target="'container'" targettype="'x-marea'" event="'set_condition'" namelist="selection" />
          </if>
        </onentry>

        <transition event="current_leg" cond="_eventdata=='hold'" target="Hold" />
        <transition event="current_leg" cond="_eventdata=='scanPoint'" target="ScanPoint" />
      </state>

    </state>
  </parallel>
</scxml>
```

C.4 Immediate Hotspot Analysis

Listing C.4 specifies the behavior of the UAS during the mission state when applying an immediate hotspot analysis strategy.

Listing C.4: SCXML immediate analysis specification.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    Definition of UAS behavior for HotSpot mission.
-->
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0"
    initialstate="ScanArea">

  <datamodel>
    <!-- What to do during mission stage:
            0: Scan Area
            1: Scan Point
            2: Hold -->
    <data id="selection" expr="0" />
    <!-- Goto leg destination -->
    <data id="goto_dest" expr="''" />
  </datamodel>

  <!-- During the mission stage, the UAS will be in one of the following states:
          Hold: Don't know what to do, just wait.
          Scan Area: Scan the area of interest.
          Scan Point: Take closer look at a potential hot spot. -->

  <state id="ScanArea">
    <datamodel>
      <data id="lat" expr="''" />
      <data id="lon" expr="''" />
      <data id="startAtCoords" expr="''" />
      <!-- Update message for scan area -->
      <data id="scanUpdate"
        xmlns:fpu='http://icarus.upc.es/schema/FlightPlanUpdate/1.1'
        xmlns:fp='http://icarus.upc.es/schema/FlightPlan/1.1'
        xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'>
        <fpu:FlightPlanUpdate xmlns="">
          <Change>
            <MainFP targetId="HotSpotPlan">
              <stages>
                <stage targetId="mission">
                  <legs>
                    <leg targetId="scanArea" xsi:type="fp:BasicScanLeg">
                      <startAt/>
                    </leg>
                  </legs>
                </stage>
              </stages>
            </MainFP>
          </Change>
        </fpu:FlightPlanUpdate>
      </data>
      <!-- Update message for scan point -->
      <data id="scanPointUpdate"
        xmlns:fpu='http://icarus.upc.es/schema/FlightPlanUpdate/1.1'
        xmlns:fp='http://icarus.upc.es/schema/FlightPlan/1.1'
        xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'>
        <fpu:FlightPlanUpdate xmlns="">
          <Change>
```

```
            <MainFP targetId="HotSpotPlan">
              <stages>
                <stage targetId="mission">
                  <legs>
                    <leg targetId="scanPoint" xsi:type="fp:ScanPointLeg">
                      <dest>
                        <coordinates />
                      </dest>
                      <course/>
                    </leg>
                  </legs>
                </stage>
              </stages>
            </MainFP>
          </Change>
        </fpu:FlightPlanUpdate>
      </data>
    </datamodel>

    <onentry>
      <!-- When done go to hold state -->
      <assign name="selection" expr="2" />
      <send target="'container'" targettype="'x-marea'" event="'set_condition'" namelist="selection" />
    </onentry>

    <transition event="hotspot">
      <!-- Get current position -->
      <assign name="lat" expr="Position.getLatitude().toString()" />
      <assign name="lon" expr="Position.getLongitude().toString()" />
      <assign name="startAtCoords" expr="lat.concat(' ').concat(lon)" />

      <!-- (1) Update scanPoint leg with hotspot position data -->
      <log expr="'HotSpot location: ' + _eventdata['coordinates']"/>
      <log expr="'HotSpot course:   ' + _eventdata['course']"/>
      <assign xmlns:fpu='http://icarus.upc.es/schema/FlightPlanUpdate/1.1'
        location="Data(scanPointUpdate, 'fpu:FlightPlanUpdate/Change/MainFP/stages/stage/legs/leg/dest/coordinates')"
        expr="_eventdata['coordinates']" />
      <assign xmlns:fpu='http://icarus.upc.es/schema/FlightPlanUpdate/1.1'
        location="Data(scanPointUpdate, 'fpu:FlightPlanUpdate/Change/MainFP/stages/stage/legs/leg/course')"
        expr="_eventdata['course']" />
      <!--  Send message to fpms -->
      <send target="'container'" targettype="'x-marea'" event="'update_cmd'" namelist="scanPointUpdate" />

      <!-- (2) Jump to scanPoint leg -->
      <assign name="selection" expr="1" />
      <send target="'container'" targettype="'x-marea'" event="'set_condition'" namelist="selection" />
      <assign name="goto_dest" expr="'scanPoint'" />
      <send target="'container'" targettype="'x-marea'" event="'goto_leg'" namelist="goto_dest" />

      <!-- (3) Update startAt parameter of scan so that flight is resumed where it was left  -->
      <assign xmlns:fpu='http://icarus.upc.es/schema/FlightPlanUpdate/1.1'
        location="Data(scanUpdate, 'fpu:FlightPlanUpdate/Change/MainFP/stages/stage/legs/leg/startAt')"
        expr="startAtCoords" />
      <!--  Send message to fpms -->
      <send target="'container'" targettype="'x-marea'" event="'update_cmd'" namelist="scanUpdate" />
    </transition>

    <transition event="current_leg" cond="_eventdata=='hold'" target="Hold">
      <!-- Update startAt parameter so that scan starts over from the beginning -->
      <assign xmlns:fpu='http://icarus.upc.es/schema/FlightPlanUpdate/1.1'
        location="Data(scanUpdate, 'fpu:FlightPlanUpdate/Change/MainFP/stages/stage/legs/leg/startAt')"
        expr="''" />
      <!--  Send message to fpms -->
      <send target="'container'" targettype="'x-marea'" event="'update_cmd'" namelist="scanUpdate" />
    </transition>

    <transition event="current_leg" cond="_eventdata=='scanPoint'" target="ScanPoint" />
  </state>

  <state id="ScanPoint">
    <onentry>
      <!-- When done go to scan state -->
      <assign name="selection" expr="0" />
      <send target="'container'" targettype="'x-marea'" event="'set_condition'" namelist="selection" />
    </onentry>

    <transition event="current_leg" cond="_eventdata=='scanArea'" target="ScanArea" />
  </state>

  <state id="Hold">
    <!-- Wait until commanded to perform scan or terminate mission -->
    <transition event="scan" target="Hold">
      <assign name="selection" expr="0" />
      <send target="'container'" targettype="'x-marea'" event="'set_condition'" namelist="selection" />
      <assign name="goto_dest" expr="'scanArea'" />
      <send target="'container'" targettype="'x-marea'" event="'goto_leg'" namelist="goto_dest" />
    </transition>

    <transition event="terminate" target="Hold">
      <assign name="goto_dest" expr="'retleg'" />
      <send target="'container'" targettype="'x-marea'" event="'goto_leg'" namelist="goto_dest" />
    </transition>

    <transition event="current_leg" cond="_eventdata=='scanArea'" target="ScanArea" />
  </state>

</scxml>
```

References

AERONAUTICAL RADIO, INC. 2008. *424-19 navigation systems data base*. 14

AIRBUS. 2002 (April). *Getting to grips with modern navigation*. Flight Operations Support & Line Assistance. 13

ANDERSON, CHRIS. 2009. *DIY Drones Ardupilot*. http://diydrones.com/profiles/blogs/ardupilot-main-page. Last visited: November 2009. 11

ARKIN, R.C. 1998. *Behavior based robotics*. The MIT Press. 17

ASMARE, E., DULAY, N., KIM, H., LUPU, E., & SLOMAN, M. 2006. Management architecture and mission specification for unmanned autonomous vehicles. *In: 1st seas dtc technical conference*. 19

BARBIER, MAGALI, & CHANTHERY, ELODIE. 2004. Autonomous mission management for unmanned aerial vehicles. *Aerospace science and technology*, **8**(4), 359 – 368. 19

BOOCH, GRADY, RUMBAUGH, JAMES, & JACOBSON, IVAR. 2005. *Unified modeling language user guide, the (2nd edition) (the addison-wesley object technology series)*. Addison-Wesley Professional. 23, 83

BRAY, TIM, PAOLI, JEAN, SPERBERG-MCQUEEN, C. M., MALER, EVE, COIS YERGEAU, FRAN¸ & COWAN, JOHN. 2006 (August). *Extensible markup language (XML) 1.1 (second edition)*. World Wide Web Consortium (W3C). http://www.w3.org/TR/xml11/. 6, 12

BRISSET, PASCAL, DROUIN, ANTOINE, GORRAZ, MICHEL, HUARD, PIERRE-SELIM, & TYLER, JEREMY. 2006 (November). *The paparazzi solution*. MAV2006. 11

CHAO, HAIYANG, CAO, YONGCAN, & CHEN, YANGQUAN. 2007. Autopilots for small fixed-wing unmanned air vehicles: A survey. *Pages 3144–3149 of: International conference on mechatronics and automation (icma)*. Harbin, China: IEEE. 9, 33

CLOUD CAP TECHNOLOGY. 2009. *Piccolo family of autopilots*. http://www.cloudcaptech.com/piccolo_system.shtm. Last visited: November 2009. 10

COX, VICTORIA, ROMANOWSKI, MICHAEL, MOHLER, GISELE, TEDFORD, ANN, & WHITLEY, PAMELA. 2009 (October). *FAA's NextGen implementation plan*. NextGen Integration and Implementation Office, FAA. 110

DAMIANOU, N., DULAY, N., LUPU, E., & SLOMAN, M. 2001. The ponder policy specification language. *In:* SPRINGER (ed), *Policies for distributed systems and networks: International workshop, policy 2001. proceedings*. 19

DEGARMO, MATTHEW T. 2004. *Issues concerning integration of unmanned aerial vehicles in civil airspace*. Tech. rept. The MITRE Corporation. 6

Doherty, P., Haslum, P., Heintz, F., Merz, T., Nyblom, P., Persson, T., & Wingman, B. 2004. A distributed architecture for autonomous unmanned aerial vehicle experimentation. *In: Proceedings of the 7th international symposium on distributed autonomous systems*. 20

Dong, Miaobo, & Sun, Zengqi. 2004 (Sept.). A behavior-based architecture for unmanned aerial vehicles. *Pages 149–155 of: Proceedings of the 2004 ieee international symposium on intelligent control*. 21

Dong, Miaobo, Chen, Ben M., Cai, Guowei, & Peng, Kemao. 2007. Development of a real-time onboard and ground station software system for a uav helicopter. *Journal of aerospace computing, information, and communication*, **4**, 933–955. 21

EM Technologies Group. 2009. *Attopilot*. `http://attopilot.com/`. Last visited: November 2009. 11

EUROCONTROL. 2003. *Guidance material for the design of terminal procedures for area navigation*. European Organisation for the Safety of Air Navigation. 6, 14

EUROCONTROL. 2009 (March). *Air traffic management master plan*. European Organisation for the Safety of Air Navigation. 110

FAA. 2005 (October). *United states standard flight inspection manual*. Departments of the Army, the Navy, and the Air Force and the Federal Aviation Administration. 67

FAA. 2007 (November). *Aviation system standards. flight inspetion operations group. flight inspetion handbook. ti 8200.52*. Federal Aviation Administration. U.S. Department of Transportation. 69, 70

FAA. 2008 (February). *Aeronautical information manual, official guide to basic flight information and atc procedures*. Federal Aviation Administration. U.S. Department of Transportation. 6, 33

FAA. 2009. *Pilot/controller glossary*. U.S. Federal Aviation Administration. 14

Freed, M., Bonasso, P., Dalal, K.M., Fitzgerald, W., & Harris, R. 2005. An architecture for intelligent management of aerial observation missions. *In: Proceedings of american institute of aeronautics and astronautics "infotech@aerospace" technical conference*. 18

Harel, D., & Politi, M. 1998. *Modeling reactive systems with statecharts: The statemate approach*. McGraw-Hill. 6, 23, 82

Heimdahl, M.P.E., Leveson, N.G., & Reese, J.D. 1998. Experiences from specifying the tcas ii requirements using rsml. *Digital avionics systems conference, 1998. proceedings., 17th dasc. the aiaa/ieee/sae*, **1**(Oct-7 Nov), C43/1–C43/8 vol.1. 23

Huang, H.M., Messina, E., & Albus, J. 2003. Autonomy level specification for intelligent autonomous vehicles: Interim progress report. *In: Proceedings of the 2003 performance metrics for intelligent systems (permis) workshop*. 5

Huang, Hui-Min, Messina, Elena, & Albus, James. 2007. *Autonomy levels for unmanned systems (alfus) framework*. Tech. rept. National Institute of Standards and Technology. 4

ICAO. 2000. *Manual on testing of radio navigation aids, doc. 8071*. 4th edition edn. 67, 68, 69, 70

Ingham, L.A. 2008. *Considerations for a roadmap for the operation of unmanned aerial vehicles (uav) in south african airspace*. Ph.D. thesis, Universiteit Stellenbosch University. 2

Kernighan, Brian W., & Ritchie, Dennis M. 1978. *The c programming language*. Prentice-Hall. 11

Lager, Torbjörn. 2010. *pyscxml - an SCXML implementation in Python*. `http://code.google.com/p/pyscxml/`. Last visited: February 2010. 88

Lazarski, Anthony J. 2002. Legal implications of the unmanned combat aerial vehicle. *Aerospace power journal*, **16:2**, 74–83. 4

Lopez, J., Royo, P., Pastor, E., Barrado, C., & Santamaria, E. 2007 (Nov.). A middleware architecture for unmanned aircraft avionics. *In: Acm/ifip/useunix 8th int. middleware conference*. 26

Mackenzie, D.C., Arkin, R.C., & Cameron, J. 1997. Multiagent mission specification and execution. *Autonomous robots*, **4**, 29–52. 21

MicroPilot. 2009. *Mp2028 series autopilots*. `http://www.micropilot.com/products-mp2028-autopilots.htm`. Last visited: November 2009. 10

NARAYAN, PRITESH P., WU, PAUL P.Y., CAMPBELL, DUNCAN A., & WALKER, RODNEY A. 2007. An intelligent control architecture for unmanned aerial systems (uas) in the national airspace system (nas). *Pages 1–11 of: Proceedings 2nd international unmanned air vehicle systems conference*. 17

NAS, MICHAEL. 2007. *Pilots by proxy: Legal issues raised by the development of unmanned aerial vehicles*. Tech. rept. UATAR (Unmanned Aircraft Technology Applications Research). 4

NAS, MICHAEL. 2008. *The changing face of the interface: An overview of uas control issues & controller certification*. Tech. rept. UATAR (Unmanned Aircraft Technology Applications Research). 5

NASA. 2006 (August). *Earth observations and the role of UAVs: A capabilities assessment. version 1.1*. NASA's Civil UAV Assessment Team. 2

NASA. 2009. *Apex*. http://ti.arc.nasa.gov/projects/apex/. Last visited: January 2009. 18

NATO. 2000 (April). *Flight testing of radio navigation systems (les essais en vol des systès de radionavigation)*. NATO Research and Technology Organisation. 70

NOKIA. 2010. *SCXML Support for the Qt State Machine Framework*. http://labs.trolltech.com/page/Projects/xml/scxml. Last visited: February 2010. 88

NYBLOM, PER. 2003. *A language translator for robotic task procedure specifications*. M.Phil. thesis, Linköpings universitet. 20

OLSON, CURTIS L. 2010. *FlightGear Flight Simulator*. http://www.flightgear.org. Last visited: April 2010. 75

OMG. 2010. *Introduction to OMG's unified modeling language (UML)*. Object Management Group (OMG). Last visited: February 2010. 23, 83

OREBÄCK, ANDERS. 2004. *A component framework for autonomous mobile robots*. Ph.D. thesis, KTH Numerisk analys och datalogi. 17

PAPARAZZI. 2010. *The paparazzi project*. http://www.recherche.enac.fr/paparazzi/wiki/index.php/Main_Page. Last visited: April 2010. 11

PASTOR, E., LOPEZ, J., & ROYO, P. 2007. UAV payload and mission control hardware/software architecture. *Ieee aerospace and electronic systems magazine*, **22**(6). 25

PROCERUS TECHNOLOGIES. 2009. *Kestrel autopilot*. http://www.procerusuav.com/productsKestrelAutopilot.php. Last visited: November 2009. 10

QVIST, IAN. 2006. Remote flight inspection of enroute facilities. *In: 14th ifis (int'l flight inspection symposium)*. 68

RAMIREZ, JORGE, BARRADO, CRISTINA, & PASTOR, ENRIC. 2009. A proposal for using uas in radio navigation aids flight inspection. *In: 47th aiaa aerospace sciences meeting*. 67, 68

ROYO, P., LOPEZ, J., PASTOR, E., & BARRADO, C. 2008. Service abstraction layer for UAV flexible application development. *In: 46th aiaa aerospace sciences meeting and exhibit*. Reno, Nevada: AIAA. 26

RTCA. 2007 (March). *DO-304: Guidance material and considerations for unmanned aircraft systems*. 2

SPITZER, CARY R. 2007. *Digital avionics handbook: elements, software and functions*. CRC Press. 15

THE APACHE SOFTWARE FOUNDATION. 2010. *Apache Commons SCXML*. http://commons.apache.org/scxml/. Last visited: February 2010. 88

THOMPSON, JEFFREY M., HEIMDAHL, MATS P. E., & MILLER, STEVEN P. 1999. Specification-based prototyping for embedded systems. *Chap. Specification-Based Prototyping for Embedded Systems, pages 163–179 of: Software engineering - esec/fse '99*. Springer Berlin / Heidelberg. 23

TRILLO, NOEL. 2009. *Rnav guidance system design for unmanned aerial vehicles*. M.Phil. thesis, Castelldefels School of Technology (EPSC). 59

UAV NAVIGATION. 2009. *AP04*. http://www.uavnavigation.com/uavprod/uavprod_01.htm. Last visited: November 2009. 10

UAVNET. 2005. *Eu civil uav roadmap*. http://www.uavnet.com. 2, 3

Ulam, P., Endo, Y., Wagner, A., & Arkin, R. 2006. *Integrated mission specification and task allocation for robot teams - part 1: Design and implementation*. Tech. rept. College of Computing, Georgia Institute of Technology. 17

UVS-International. 2009. *2009/2010 uas yearbook - uas: The global perspective*. Blyenburgh & Co. 2

W3C. 2009 (October). *State Chart XML (SCXML) state machine notation for control abstraction (W3C Working Draft)*. World Wide Web Consortium (W3C). `http://www.w3.org/TR/scxml/`. 6, 23, 81, 86

Wede, Thomas. 2006. The future of the flight inspection world, a cristall ball look into changes ahead, based on current trends and development. *In: 14th ifis (int'l flight inspection symposium)*. 68